IADESS: Infraestructura para la Auto-gestión Descentralizada de Sistemas Distribuidos

Sistemas Informáticos. 2007-2008

Facultad de Informática

Universidad Complutense de Madrid

José María Fernández de Alba López de Pablo <jmfernandezdalba@gmail.com>
Carlos Rodríguez Fernández <carlosrodriguez@computer.org>
Damiano Spina Valenti <damianospina@gmail.com>

Dirigido por:
Francisco Garijo Mazario <fgarijo@tid.es>
Juan Pavón Mestras <jpavon@fdi.ucm.es>

Patrocinado por:
Telefónica I+D

Resumen. La auto-gestión en sistemas distribuidos es una manera de afrontar la creciente complejidad de éstos hoy en día. IBM propuso el concepto de Autonomic Computing, junto con un modelo de arquitectura jerárquica para la auto-gestión. Sin embargo, en una jerarquía existen elementos críticos para el funcionamiento de todo el sistema, lo cual, en un entorno distribuido, merma la robustez y la extensibilidad de éste.
En este trabajo se desarrolla otra modelo arquitectónico para sistemas auto-gestionados, pero considerando una estructura de control no jerarquizada, descentralizada y cooperativa, para evitar tener un punto crítico de fallo. La auto-gestión de un sistema se alcanza mediante la cooperación de sus componentes transformados en auto-gestionados.
La arquitectura se sustenta sobre una infraestructura software que permite alcanzar otro de los objetivos del trabajo: poder modificar un sistema distribuido existente de una manera sencilla, mecánica y rápida. La transformación de un sistema puede ir haciéndose incrementalmente sin trastornar el funcionamiento del mismo, facilitando su evolución paulatina hacia sistemas auto-gestionados.
Finalmente, se pretende validar la propuesta arquitectónica descentralizada de sistemas auto-gestionados sobre un sistema basado en servicios existente. La implementación de la arquitectura permitirá comprobar si efectivamente el sistema resultante es más flexible y tolerante a fallos que con una solución jerarquizada, y estimar el esfuerzo requerido para dotar de capacidad de auto-gestión a un sistema existente.
Palabras claves: auto-gestión, computación autónoma, sistemas multi-agentes, redes sociales.

Abstract. Self-management in distributed systems is a way to cope with the growing complexity of these ones today, and its support in existing systems requires a transformation in their architectures. This work presents a decentralized model for the implementation of self-management capabilities, which also has the advantage of avoiding the single point of failure (SPOF) issue, providing more robustness to the management system. The proposed architecture has been validated in a real distributed application.
Key words: self-management, autonomic computing, multi-agent system, social network

Tabla de contenido

1. Introducción

1.1. Planteamiento del Problema

Los sistemas informáticos distribuidos son cada vez más complejos. Cuanto más complejo es un sistema, más puede fallar y mayor es el coste de su mantenimiento.

El concepto de **Autonomic Computing** surgió de la mano de IBM[14] con el objetivo de dejar el mantenimiento de los sistemas en manos de los propios sistemas (*auto-gestión*), liberando a los administradores de las tareas de bajo nivel para llevar a cabo tan sólo las tareas de alto nivel y ofreciendo una disponibilidad del servicio 24/7. La auto-gestión se divide en cuatro sub-objetivos a alcanzar: auto-configuración, auto-optimización, auto-reparación y auto-protección.

En su *Architectural Blueprint* [9], IBM plantea una arquitectura jerarquizada para llevar a cabo el Autonomic Computing. Sin embargo, en una jerarquía existen piezas clave para el funcionamiento de todo el sistema, lo cual, en un entorno distribuido, merma la robustez y la extensibilidad de éste.

En este trabajo se explora otra vía que intenta conseguir la auto-gestión, pero considerando una estructura de control no jerarquizada, descentralizada y cooperativa, para evitar tener un punto crítico de fallo (SPOF, Single Point of Failure). La auto-gestión de un sistema se alcanza mediante la cooperación de sus componentes transformados en auto-gestionados.

La transformación de un componente da como resultado un componente auto-gestionado que desacopla la funcionalidad de servicio o de uso con la funcionalidad de gestión.

1.2. Objetivos del Proyecto

El objetivo de este trabajo es desarrollar y validar un modelo arquitectónico no jerárquico , descentralizado, autónomo y cooperativo en el cual el control de la auto-gestión reside de modo distribuido en cada uno de los componentes del sistema y en su capacidad de coordinación. Este modelo está orientado a sistemas basados en servicios web.

Este planteamiento contrasta con el modo de ver la auto-gestión de forma centralizada y jerárquica [9]. La auto-gestión se lleva a cabo por cada componente de forma descentralizada, autónoma y cooperativa. Con esta visión distribuida se resuelven gran parte de los problemas de robustez y extensibilidad mencionados.

La arquitectura se sustenta sobre una infraestructura software que permite alcanzar otro de los objetivos del trabajo: poder modificar un sistema distribuido existente de una manera sencilla, mecánica y rápida. La transformación de un sistema puede ir haciéndose incrementalmente sin trastornar el funcionamiento del mismo, facilitando su evolución paulatina hacia sistemas auto-gestionados.

Finalmente, se pretende validar la propuesta arquitectural descentralizada de sistemas auto-gestionados sobre un sistema basado en servicios existente. La implementación de la arquitectura permitirá comprobar si efectivamente el sistema resultante es más flexible y tolerante a fallos que con una solución jerarquizada, y estimar el esfuerzo requerido para dotar de capacidad de auto-gestión a un sistema existente.

1.3. Planteamiento del Trabajo

Para conseguir abordar los objetivos propuestos, el trabajo se ha organizado de la siguiente manera:

1. **Desarrollo de un sistema distribuido prototipo basado en servicios**: Con el propósito de familiarizar a los desarrolladores con las tecnologías y problemas implicados en estos sistemas, para así dotarlos de una mayor comprensión a la hora de describir la arquitectura de auto-gestión.
2. **Análisis de la estrategia para conseguir la auto-reparación de un sistema distribuido basado en servicios existente**: Ha consistido en aplicar una transformación a cada componente del mismo, con la intención de que adquiera todas las capacidades necesarias tanto para comunicarse con los otros componentes del sistema, como para afectar a su propio estado, de tal manera que consiga detectar y hacer frente a eventuales errores en su

funcionamiento.

3. **Definición de la arquitectura de la infraestructura de auto-gestión**: En el caso de estudio sólo se ha implementado la auto-reparación, que permite validar los elementos esenciales y funcionamiento de la arquitectura. El objetivo actual ha sido la definición y validación de una arquitectura abierta para la posterior aplicación del resto de objetivos de auto-gestión.
4. **Desarrollo de una herramienta auxiliar de monitorización**: Con el fin de obtener resultados concretos sobre rendimiento y coste para obtener conclusiones sobre la viabilidad de la solución.

Se han dejado como trabajo futuro otros objetivos de auto-gestión, como la auto-configuración y la auto-optimización, que requerirían del desarrollo de estrategias que están fuera del ámbito de este trabajo.

1.4. Estructura de la Memoria

El capítulo 2 ofrece una revisión del estado del arte, en la que se describen los conceptos que aparecen en la literatura sobre este tema, así como las diferentes ideas y planteamientos ya existentes al respecto. A continuación, en el capítulo 3 se describen los principios y elementos fundamentales de la arquitectura de sistema auto-gestionado propuesto, analizando las posibles ventajas. El capítulo 4 presenta la arquitectura detallada, describiendo diseño e implementación, tanto de la infraestructura como de las herramientas auxiliares. Esta arquitectura se ha especializado para sistemas basados en servicios Web.

Para validar la arquitectura de sistemas auto-gestionados se ha desarrollado un prototipo de aplicación de red social, utilizando tecnologías de servicios Web, que se describe en el capítulo 5. Esta aplicación ha permitido evaluar la eficacia de la infraestructura desarrollada y su proceso de implantación. Estos resultados se discuten en la segunda parte de dicho capítulo. Finalmente, el capítulo 6 resume las principales aportaciones de la arquitectura propuesta en el ámbito de los sistemas auto-gestionados, e identifica alternativas y líneas de trabajo futuro.

En los anexos, se proporciona documentación complementaria sobre otros aspectos del proyecto: descripciones más detalladas del sistema de prueba y del framework utilizados ICARO-T 4.1 [24] (ver Anexo B), y una copia de un artículo sobre el trabajo aceptado para su publicación en el International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI'08). El anexo C muestra una copia del correo electrónico de aceptación del artículo y el enlace a la página web del congreso donde se indica su publicación en Springer Verlag.

2. Revisión del Estado del Arte

2.1. Introducción

Los conceptos principales de la auto-gestión han sido desarrollados inicialmente por IBM [14] y la arquitectura propuesta para la auto-gestión de sistemas que más a trascendido es la que plantearon inicialmente [9]. Estos trabajos sirven de base para exponer en el apartado 2.1, de forma resumida, qué es la auto-gestión y sus cuatro aspectos principales.

A continuación, en el apartado 2.2, se describe la arquitectura planteada por IBM, cuyo análisis ha permitido identificar una serie de cuestiones que se abordan en este trabajo.

Finalmente, en el apartado 2.3 se describen los conceptos que maneja el estándar WSDM [18, 19] debido a que está fuertemente relacionado con la gestión de sistemas a través de la tecnología Web Services, ya que estandariza las interfaces y los protocolos, y define conceptos relacionados con tal fin.

2.2. La Auto-gestión

La esencia de la computación autónoma se basa en el concepto de auto-gestión. Su objetivo es de librar al administrador de los detalles de la gestión del sistema y proveer un sistema al usuario con disponibilidad 24/7.

La auto-gestión se compone de cuatros aspectos que son los siguientes: auto-configuración, auto-optimización, auto-reparación y auto-protección, los cuales han sido introducidos en el capítulo anterior. Cada uno de estos aspectos son a su vez objetivos a alcanzar, para dotar a un sistema de la autonomía necesaria como para liberar de esfuerzo a los administradores, consiguiendo así una mayor calidad del servicio.

Los objetivos de cada uno de los aspectos de la auto-gestión son los siguientes:

> **Auto-configuración:** Por lo general, la instalación y configuración de grandes sistemas consume mucho tiempo y esfuerzo. La auto-configuración consiste en que partiendo de políticas de alto nivel definidas por los administradores el sistema se configura completamente, descubriendo e infiriendo el resto de parámetros de configuración de bajo nivel.
>
> **Auto-optimización:** En los sistemas suelen haber maneras de ajustar ciertos parámetros que logran optimizar su funcionamiento, pero el ajuste de dichos parámetros de manera manual suele ser costoso por el esfuerzo que conlleva. La auto-optimización consiste en automatizar esta operación, es decir, que el mismo sistema descubre las posibilidades de optimizarse a si mismo y realiza las operaciones necesarias para lograrlo.
>
> **Auto-reparación:** Los errores en sistemas complejos suelen llevar mucho tiempo de trabajo para descubrirlos y para repararlos. La auto-reparación consiste en la capacidad del sistema de detectar las causas de los fallos, y determinar y ejecutar las acciones necesarias para su reparación.
>
> **Auto-protección:** La protección ante ataques y errores en cascada en los sistemas suele ser manual. La auto-protección es la capacidad del sistema de detectar estos ataques y errores en cascada y protegerse de ellos.

2.3. Arquitectura para la Auto-gestión planteada por IBM

En el paper *An architectural blueprint for autonomic computing* [9], de 2006, IBM propone una arquitectura para dar soporte al *Autonomic Computing*. Según esta arquitectura, un sistema autonómico estaría compuesto por un conjunto de bloques o componentes organizados en capas de tal manera que el comportamiento individual de cada bloque, y las comunicaciones entre ellos, producen un comportamiento global de auto-gestión. Estos bloques con capacidad de decisión y actuación autónoma, pero también con capacidad de coordinación entre sí, forman un *sistema multi-agente* [2].

Las capas de un sistema de este tipo están organizadas tal como se muestra en la Ilustración 1, y que se describen a continuación, comenzando por la capa más “baja” desde el punto de vista de la abstracción:

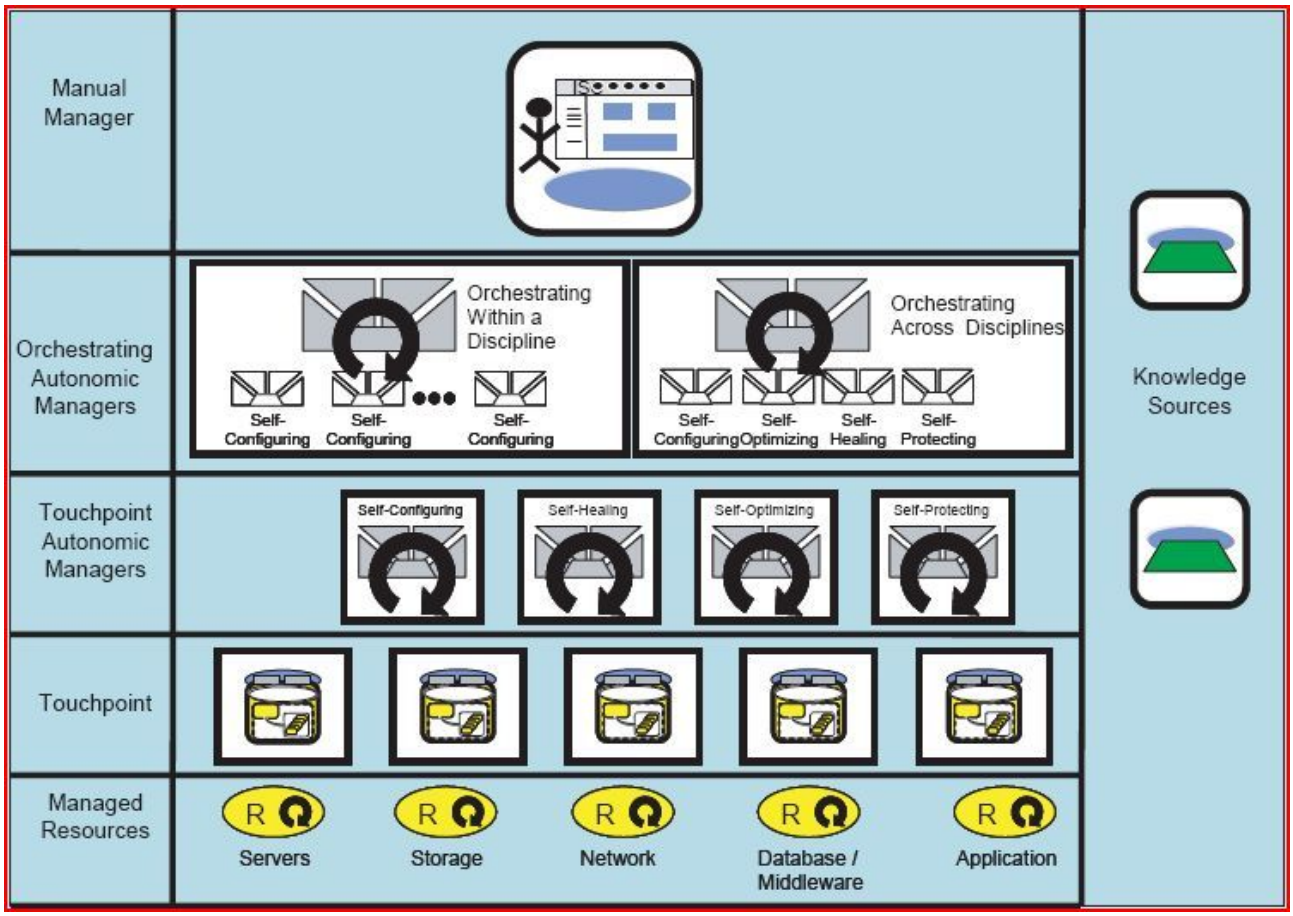

Ilustración 1: Esquema arquitectónico descrito en el paper de IBM [12]

Managed Resources (Recurso Gestionado)

En esta capa se sitúan todos los recursos *gestionables* de un sistema. Un recurso gestionable es cualquiera que los desarrolladores quieran contemplar como tal, por ejemplo un servidor, una fuente de almacenamiento, o incluso la propia red. Como casi cualquier elemento del sistema puede considerarse un recurso, se da la libertad para elegir cuáles de ellos se incluyen en esta capa y cuales se dejan simplemente fuera del alcance de la auto-gestión.

Manageability Endpoints o Touchpoints (Puntos de Acceso para la Gestión)

En esta capa se sitúan los componentes asociados a cada uno de los recursos de la capa inferior que dan soporte a diferentes interfaces de gestión según las capacidades de dichos recursos. La implementación de estas interfaces permite el acceso por parte de los *Autonomic Manager* al estado de los recursos, a la suscripción de este a sus cambios y la modificación de parámetros de gestionabilidad.

Las comunicaciones a partir de aquí se llevan a cabo mediante un Bus de Servicios de Empresa middleware. Por ejemplo, en el caso de Web Services el estándar de gestionabilidad será WSDM.

Touchpoint Autonomic Managers (Gestores de Autonomía de los Puntos de Accesos para la Gestión)

Los Autonomic Managers se implementan como agentes software, asociados a cada uno de los terminales de gestionabilidad de la capa inferior. Estos Autonomic Manager se suscriben a los cambios de estado y eventos de sus respectivos recursos por medio de los terminales de gestionabilidad y, según el caso, pueden suscribirse también a los cambios de estado de otro recurso externo al suyo propio del que dependan.

Cada instancia de Autonomic Manager tiene asociada un conjunto de *directivas*, las cuales son especificadas por agentes de un nivel "superior" a ellos, que pueden ser Autonomic Managers de Orquestación o Managers Manuales controlados por usuarios administradores. Estas directivas definen los objetivos a alcanzar por los Autonomic Managers.

Orchestrating Autonomic Managers (Gestores de Autonomía para Orquestación)

En esta capa se sitúa la implementación de los Autonomic Managers de Orquestación, los cuales son Autonomic Managers que siguen directivas de más alto nivel y cuyos recursos gestionados son a su vez otros Autonomic Managers. Su tarea consiste en coordinarlos para llevar a cabo acciones conjuntas. En esta capa se situaría el Autonomic Manager que gestiona el sistema completo.

Manual Manager (Gestores Manuales)

En esta capa están las implementaciones de las consolas de gestión que permiten a los usuarios operadores monitorizar y dar órdenes al sistema. Desde aquí se tiene control sobre el sistema mediante el Bus de Servicios de Empresa. Así, se pueden especificar las directivas a asignar al Autonomic Manager del sistema, e incluso, si se quiere tener más control, se puede permitir el acceso a los Autonomic Manager de más bajo nivel o directamente a los recursos. Además, este componente puede estar suscrito a los eventos del sistema para informar al usuario de qué ocurre en cada momento. En el caso de que las directivas especificadas no permitan a los Autonomic Manager llevar a cabo planes de acción por su propia cuenta, éste sería el componente sobre el que los Autonomic Manager delegarían la decisión, de tal manera que el usuario pueda decidir lo que hacer.

Knowledge Sources (Fuente de Conocimiento)

Las fuentes de conocimiento son accesible por todos los Autonomic Managers y contienen el conocimiento necesario para llevar a cabo la autogestión del sistema. Este conocimiento puede incluir: topología del sistema, logs y métricas globales, directivas globales, etc.

El modelo arquitectónico estructurado en capas de IBM es fácilmente comprensible, y se apoya en una gestión de los recursos con el estándar WSDM, lo cual permite la extensibilidad de un sistema para interconectarse con otros sistemas de manera casi inmediata. Además, el control de la auto-gestión descansa sobre los componentes Autonomic Manager, externos a los propios recursos auto-gestionados. Es, por tanto, un enfoque apropiado desde el punto de vista de la complejidad y mantenibilidad. En el siguiente apartado se detallan los puntos más significativos del estándar.

2.4. El Estándar WSDM

Web Services Distributed Management (WSDM) [18,19] es un estándar que describe un conjunto de protocolos e interfaces que permiten la gestión por medio de Servicios Web de una colección de recursos en un sistema.

Según el estándar, cada recurso gestionable expone una interfaz Web Services que lo representa llamado WS-Resource [20]. Esta interfaz debe ser referenciada por un Endpoint-Reference (EPR) que sigue el estándar WS-Addressing [25] para poder ser localizada.

Se pueden llevar a cabo tres tipos de interacciones sobre un WS-Resource:

- Recuperar información del recurso.
- Cambiar el estado del recurso cambiando su información de gestión.
- Suscribirse a un determinado tema relacionado con el recurso.

Los elementos de la arquitectura son los siguientes:

Resource Property Document (Documentos de las Propiedades del Recurso)

Describe la representación XML de un recurso gestionable. Este documento es referenciado en el WSDL portType que describe las operaciones del WS-Resource. Representa una composición lógica de propiedades del recurso para ofrecer una visión de su estado.

Manageability Capabilities (Capacidades de Gestión)

Son un conjunto de propiedades, operaciones y eventos que permiten que un recurso sea gestionado de alguna forma. Se especifican en el RPD de cada recurso.

Las “capacidades” definidas en WSDM son las siguientes:

Identidad: expone el *ResourceID* del recurso, un ID único que lo identifica unívocamente. Todos los recursos deben tener esta capacidad.

Características de Gestionabilidad: expone una lista de capacidades de gestionabilidad que soporta el recurso. Pueden ser capacidades WSDM o específicas del recurso.

Propiedades de Correlación: expone una lista de propiedades que son útiles para determinar si dos *ResourceIDs* se refieren en realidad al mismo recurso.

Descripción: expone la captura, la descripción y la versión del recurso.

Estado: expone el estado actual y la última transición del recurso. Puede que cada recurso tenga su propio modelo de estados. Este modelo debe ser visible por el consumidor del recurso. En el caso de los Web Services se ha utilizado el estándar "Web Service Management: Service Life Cycle"

Estado de Operatividad: expone la "salud" del recurso desde un punto de vista de la operatividad. La propiedad tiene los valores: Disponible, Parcialmente Disponible, No Disponible y Desconocido.

Métricas: expone métricas relativas al rendimiento y operación del recurso. WSDM define algunas, pero pueden definirse nuevas.

Configuración: expone propiedades que el consumidor puede cambiar para modificar el comportamiento del recurso.

Management Events (Eventos de Gestión)

Son notificaciones asíncronas que denotan un cambio significativo del estado de un recurso. Cada capacidad está asociada con un tema sobre el que se pueden generar notificaciones y un consumidor puede elegir el tema, los filtros y la forma de envío de las notificaciones cuando efectúa la suscripción. Éstos mensajes tienen un formato específico descrito en WSDM Event Format (WEF).

Message Exchange Patterns (Patrones de Intercambio de Mensajes)

WSDM define un conjunto de patrones de intercambio de mensajes (MEPs) que describen cómo deben interactuar un recurso y un consumidor y se dividen según los tres tipos de interacciones que se pueden realizar sobre un recurso: Peticiones de Información de Propiedades, Establecimiento de Valores de Propiedades, y Suscripciones y Notificaciones a Eventos.

El uso de este estándar no está supeditado a un tipo concreto de arquitectura. Actualmente existen implementaciones de este estándar siguiendo una arquitectura basada en la propuesta anteriormente descrita. Tal es el caso de [21] donde se describe la experiencia de implementar Autonomic Computing en un sistema distribuido basado en la Tecnología de Servicios Web.

2.5. Arquitecturas para dominios concretos

Existen también otras arquitecturas de auto-gestión, aunque centradas en el dominio al que se aplican. Tales son los casos de:

- RISE [15]: En esta arquitectura se tratan aspectos de la auto-gestión muy particulares, en concreto la auto-configuración y la auto-reparación de imágenes remotas de sistemas.
- Adaptación en Workflow [16]: Se enfoca en tratar la adaptación de los procesos de flujos de trabajo (workflow) ante cambios del entorno con el fin de mejorar el rendimiento.
- Auto-reparación en la Computación ubicua [23]: Enfocado en la creación de una capa para la auto-reparación de aplicaciones para computación ubicua.

2.6. Conclusiones

El modelo de IBM tiene como ventajas su estructuración en capas y el apoyarse en estándares para la gestión de los componentes. Sin embargo, existe un grupo de componentes en las capas superiores que controlan la gestión de todo el sistema, lo que los convierte en componentes críticos *SPOF* (Single Point Of Failure): si alguno de estos componentes deja de estar operativo, la auto-gestión del todo el sistema puede verse mermada. Aparte de esto, el hecho de que la gestión de todos los recursos se haga de manera externa por medio del estándar WSDM, el cual se basa en Web

Services, obliga a establecer una conexión de red con el recurso, creando otra vez una forma de aislar los componentes gestionados por fallos de red, haciéndolos inaccesibles y comprometiendo la auto-gestión de todo el sistema. Este es precisamente uno de los aspectos que se abordan en este trabajo, para lo cual se plantea una arquitectura más descentralizada y en la que la auto-gestión se acopla a los componentes gestionados.

Con respecto al estándar WSDM, muchos de los conceptos tratados son usados en la propuesta de este proyecto debido a su generalidad y utilidad en este dominio. Estos son: Estado, Estado de Operatividad, Eventos de Gestión y algunos MEPs. A pesar de ello, no se ha adoptado el uso del mismo por la complejidad, extensión excesivas y la falta de una implementación madura, lo que lo hacía incompatible con las restricciones de tiempo.

De los dominios concretos donde se aplicó la auto-gestión, resulta interesante para este trabajo la propuesta [23], la cual se enfoca en la creación de una capa software reutilizable, que se instalaría en todos los dispositivos, para la auto-gestión de aplicaciones en la computación ubicua. Esta propuesta persigue una solución parecida a la de este trabajo, salvo que aborda más directamente la problemática de las aplicaciones distribuidas en general.

El interés actual en todas estas ideas se intuye por la reciente creación del grupo de trabajo Self-Management del European Telecommunications Standards Institute (ETSI) y la aparición de workshops y conferencias en los temas de Autonomic-Computing y Auto-gestión como son:

- 1ST INTERNATIONAL WORKSHOP ON AGENTS FOR AUTONOMIC COMPUTING (AAC 2008) June 6, 2008 in Chicago, http://www.iids.org/aac-2008/
- 2008 SIWN International Conference on Self-organization and Self-management in Computing and Communications, Glasgow, UK, 22-24 July 2008 http://siwn.org.uk/2008/
- The Second International Conference on Autonomic Computing and Communication Systems, September 23-25, 2008, TURIN, http://www.autonomics.eu/

3. Propuesta de Arquitectura de Sistemas Auto-gestionados

3.1. Planteamiento de la Propuesta

Frente a las arquitecturas jerárquicas existentes, como la descrita de IBM, este trabajo propone una arquitectura completamente distribuida, que resuelve el problema del SPOF (Single Point of Failure). En esta propuesta, la auto-gestión de todo el sistema se consigue proporcionando auto-gestión a cada uno de sus componentes. Al no existir una jerarquía de control centralizado, cada componente ha de estar capacitado para arreglárselas por sí mismo en su entorno, y por tanto no depende de otros componentes para su autogestión. De esta manera, no ocurre que el fallo en un componente o un pequeño grupo de ellos, merme la autogestión de todo el sistema.

La forma de conseguir que un sistema trabaje bajo dicha arquitectura consiste en el diseño e implementación de una infraestructura que consiga transformar sistemas basados en Web Services con el fin de convertirlos en auto-gestionados. Esta transformación se puede realizar aplicando la infraestructura de autogestión a cada uno de los componentes del sistema de manera independiente, obteniendo así componentes auto-gestionados. La autogestión a nivel de sistema se obtiene mediante la cooperación entre sus componentes auto-gestionados. Esta cooperación puede tener muchas formas y podría estructurarse de forma jerárquica o mediante relaciones peer-to-peer, dependiendo de las necesidades de la aplicación y su entorno. En la Ilustración 2 se muestra un ejemplo de transformación basado en el caso de estudio.

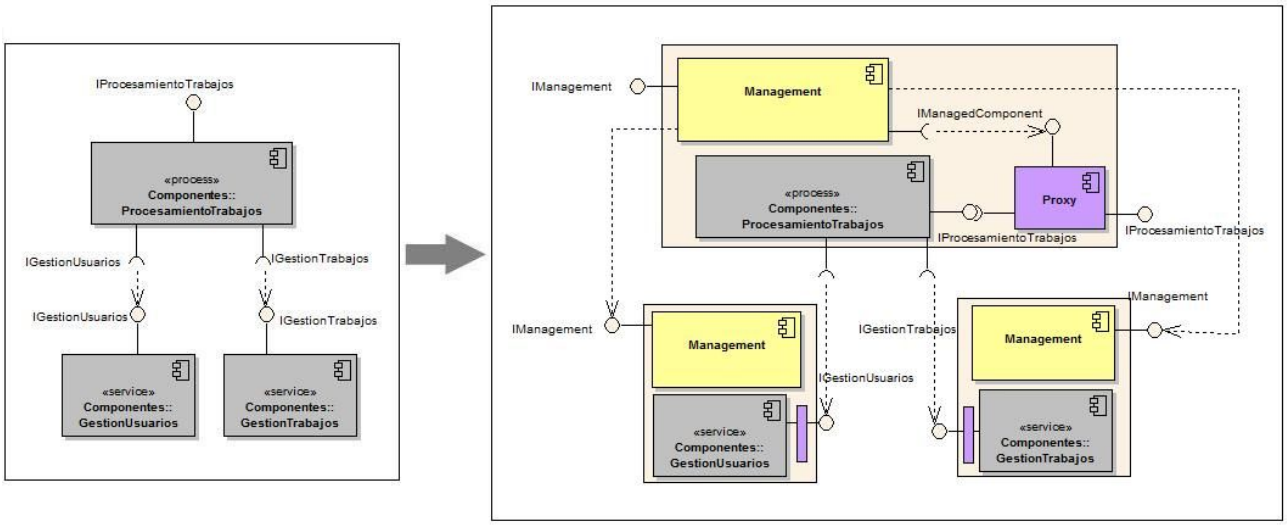

Ilustración 2: Transformación de un sistema en auto-gestionado

De esta manera, un sistema puede convertirse en auto-gestionado de forma totalmente distribuida, sin necesidad de tener una base de conocimiento centralizada, como en el caso de la propuesta de IBM descrita en la sección ,que sería un punto crítico del sistema. En la sección 4.1.4 se describen la información necesaria (interfaces, ficheros de configuració, etc.) para transformar un sistema existente en auto-gestionado usando la Infraestructura IADESS.

Cuando el sistema falla es porque un componente del sistema falla. La auto-gestión de dicho componente se encarga de detectar el fallo, informar a los componentes que de él dependen e intentar resolver la incidencia. La información se propaga al resto de componentes, para así tomar medidas de prevención (p.e. deshabilitar parte de su funcionalidad) o de reparación (p.e. buscar otro servicio que supla la dependencia).

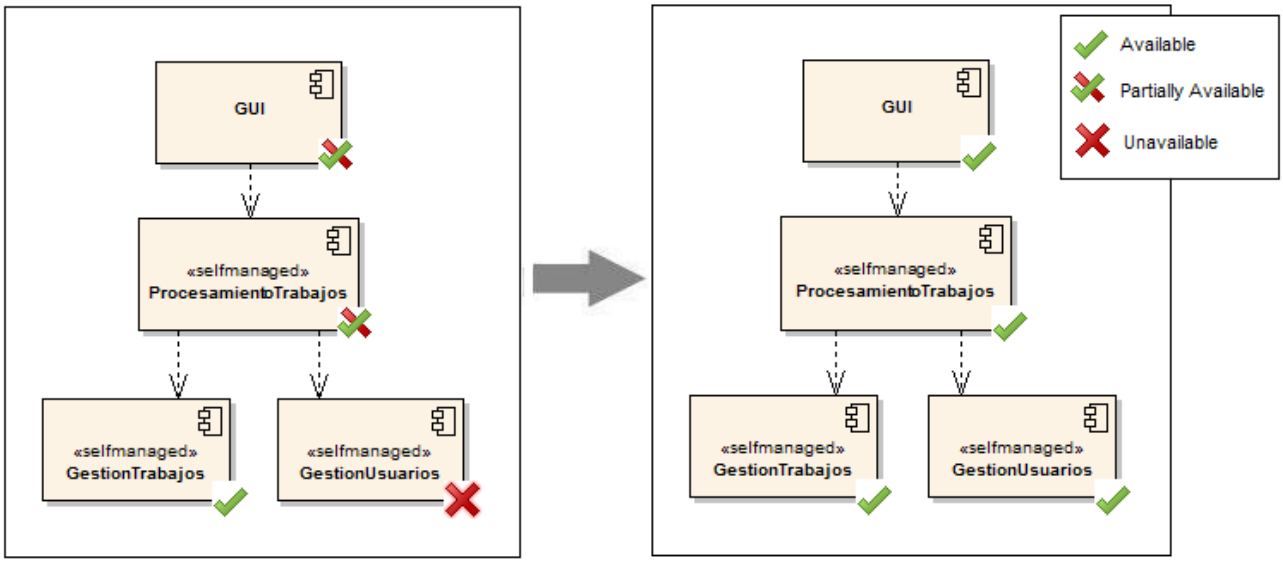

Ilustración 3: Auto-reparación del sistema

En la Ilustración 3 se muestra un ejemplo de auto-reparación del sistema. El componente *GestionUsuarios* falla y publica su nuevo estado (*no disponible*). El componente *ProcesamientoTrabajos* que depende de *GestionUsuarios* cambia su estado a *parcialmente disponible* y lo notifica al componente de presentación *GUI*. A su vez, *ProcesamientoTrabajos* intenta vincularse con un nuevo servicio que proporcione la misma funcionalidad que *GestionUsuarios*. Lo mismo ocurre con el componente *GUI*, que cambia su estado a *parcialmente disponible* e intenta buscar un componente que le proporcione la funcionalidad que le falta. Una vez que el componente *GestionUsuarios* se auto-repara, el resto de componentes pueden nuevamente proveer toda su funcionalidad y pasan al estado *disponible*.

A continuación se describirá brevemente la arquitectura interna de un componente auto-gestionado que se obtiene al aplicar la infraestructura de autogestión.

3.2. Arquitectura de un componente auto-gestionado

Cada componente tiene partes internas como pueden ser ficheros, librerías... y posiblemente dependencias con otros componentes y servidores. Estos últimos serán monitorizados, analizados y controlados para poder dar las capacidades de autogestión a cada componente y así al sistema global.

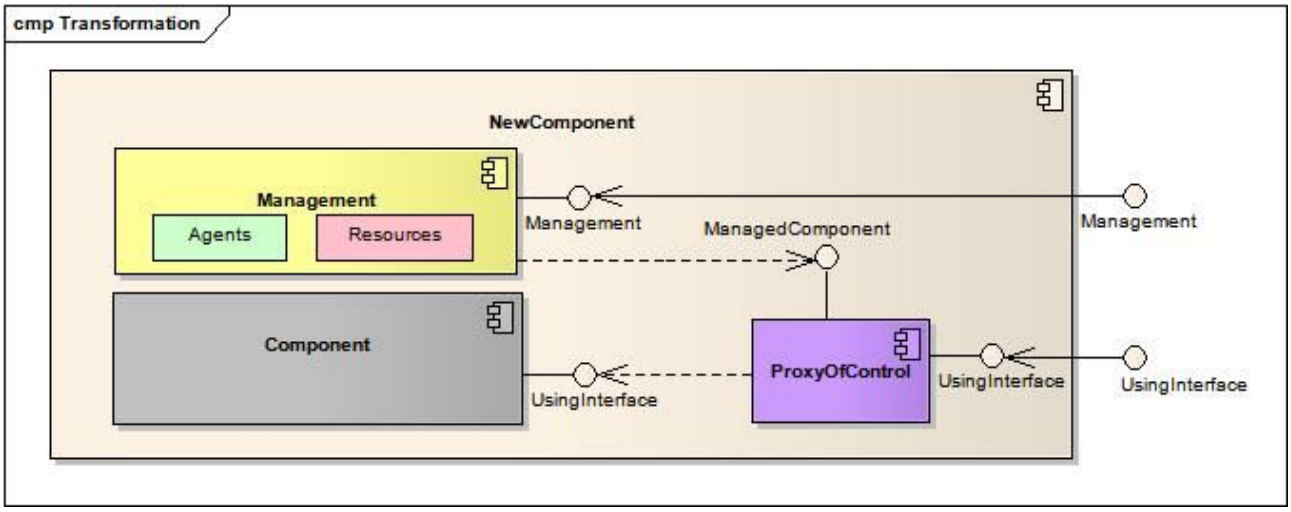

Ilustración 4: Arquitectura de un componente autogestionado

En la Ilustración 4 se muestra la estructura interna de un componente de servicio. “Component” es el componente

original que provee las operaciones lógicas del servicio. “Management” y “ProxyOfControl” son los componentes que implementan las capacidades de autogestión. Éstos últimos se añaden al componente original para construir “NewComponent”, que es el componente que provee las operaciones lógicas originales y las funcionalidades de autogestión.

El componente “Management” se compone de agentes, recursos y un modelo, los cuales se describen en las secciones 3.4, 3.5 y 3.6 respectivamente. La interfaz Management ofrece al resto de componentes autogestionados operaciones para conocer su “status” operacional y operaciones de notificación y suscripción que utilizan para coordinarse.

El componente "ProxyOfControl" controla el acceso al componente gestionado, evitando posibles malos usos en estados inapropiados, y provee información sobre el estado del componente auto-gestionado capturando las posibles excepciones técnicas que se pudieran lanzar. Este componente ha sido diseñado utilizando el patrón de diseño Estado (State) [7].

Para poder saber si un componente está funcionando correctamente, es decir, si su estado es correcto, hace falta monitorizar y analizar el comportamiento esperado de los componentes de los que depende. En otras palabras, el estado del componente es dado por el estado de los componentes internos y externos de los cuales éste depende. Si en un determinado momento el estado del componente no es correcto, entonces se toman medidas para intentar resolver las incidencias que causan el encontrarse en dicho estado.

En la Ilustración 5 muestra en con un diagrama de colaboración el siguiente escenario de reparación:

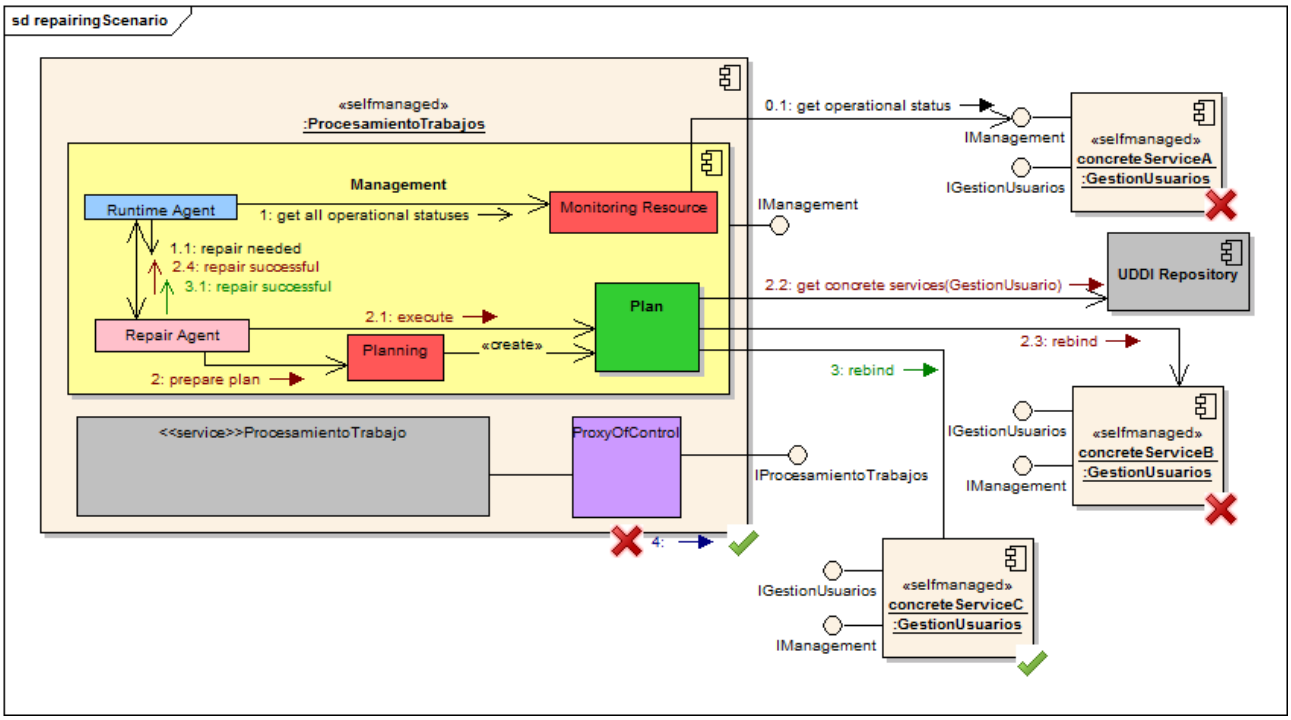

Ilustración 5: Escenario de reparación. Vinculación a un servicio concreto

1. El componente auto-gestionado *ProcesamientoTrabajos* está vinculado con el servicio concreto *concreteServiceA* que ofrece el servicio *GestionUsuario.*
2. El agente Runtime del componente *ProcesamientoTrabajo* detecta que *concreteServiceA* puede no estar funcionando correctamente, mediante un recurso de monitorización (mensaje 1).
3. El agente Runtime añade concreteServiceA a su lista negra de servicios concretos que ofrecen el servicio GestionUsuario y pide al agente Manager que se cree un agente Repair (este paso no se muestra en el diagrama para simplificarlo)
4. El agente Runtime avisa al agente Repair de que es necesaria una reparación (mensaje 1.1).
5. El agente Repair planifica un plan y lo ejecuta (mensajes 2 y 2.1).

6. La ejecución del plan consiste en:
    - obtener del repositorio UDDI la lista de servicios concretos que ofrecen el servicio *GestionUsuarios* (mensaje 2.2)
    - elegir un servicio concreto que no se encuentre en la lista negra. Se elige *concreteServiceB.*
    - vincular con *concreteServiceB* (mensaje 2.3).
7. El agente Repair notifica al agente Runtime de que el plan de reparación se ha ejecutado correctamente (mensaje 2.4).
8. Pasado un tiempo, el servicio *concreteServiceB* cambia de estado a *no disponible*.
9. El agente Runtime obtiene el estado de *concreteServiceB* nuevamente mediante el mismo recurso de monitorización. Se repiten de forma análoga los pasos 3,4 y 5 (estos pasos también han sido omitidos en el diagrama).
10. La ejecución del nuevo plan consiste en obtener nuevamente la lista de servicios concretos y vincular con concreteServiceC (mensaje 3).
11. El agente Repair notifica al agente Runtime (mensaje 3.1).
12. El agente Runtime detecta que los servicios a los que está vinculado se encuentran en estado *disponible* y cambia su estado de *no disponible* a *disponible* (mensaje 4).

A continuación se describirán los conceptos de dependencias internas y externas, que son las dependencias de un componente auto-gestionado.

3.3. Dependencias de un componente auto-gestionado

Para conseguir la funcionalidad de autogestión, es necesario tener un modelo conceptual del dominio, representando explícitamente las dependencias entre componentes [6]. La Ilustración 6 describe el modelo de dependencias que comparten los componentes gestionados.

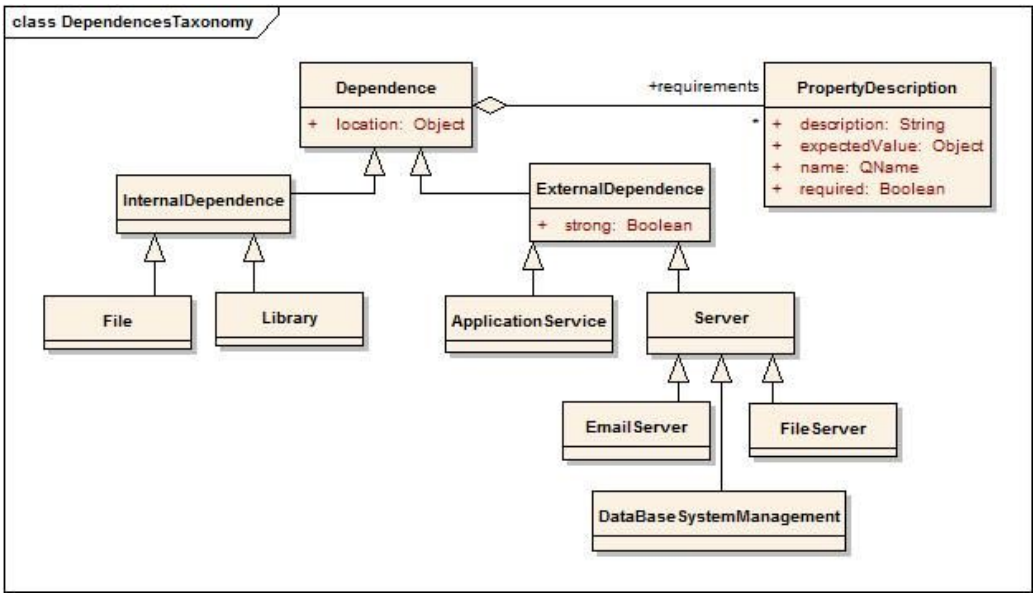

Ilustración 6: Modelo de dependencias

Un componente gestionado puede tener dos tipos de dependencias:

- **dependencia interna:** es una relación de uso con una entidad computacional, como puede ser un fichero o una librería.
- **dependencia externa:** es una relación de uso con un servidor, p.e. un servidor de correo, un servidor de base

de datos, un servidor de ficheros, o con otro servicio de aplicación. En este caso, la dependencia externa de un servicio de aplicación (Application Service) se refiere a un servicio abstracto, es decir, una interfaz requerida, donde la vinculación con un servicio concreto se realiza en tiempo de ejecución.

Todas la dependencias tienen una localización (location). Este atributo indica cómo se accede al componente que suple dicha dependencia, para usarlo o para monitorizarlo. Para los servicios de aplicación, la localización se refiere a la interfaz requerida y a la localización del servicio de registro donde poder obtener un servicio concreto que supla la dependencia.

Además, la dependencia tiene un conjunto de requisitos que definen las propiedades que deben satisfacerse por el componente que suple la dependencia. Ejemplos de propiedades pueden ser: permiso de lectura y escritura de un fichero, validación de un XML, tiempo de respuesta, disponibilidad, etc. En la sección 4.1.5 se describen con más detalle estas propiedades.

3.4. Agentes

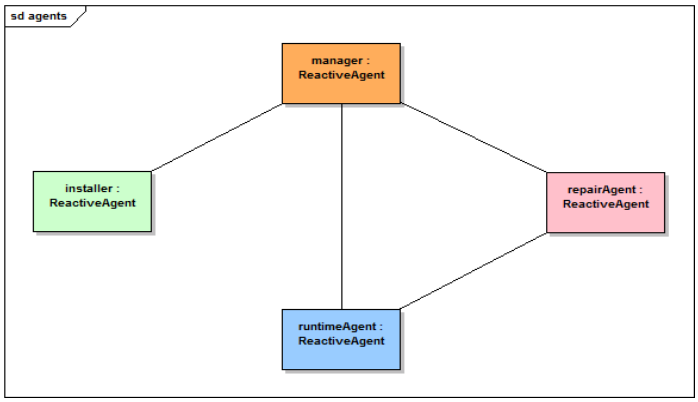

Ilustración 7: Agentes

El control lógico ha sido diseñado utilizando el paradigma Multi-Agente [2], y ha sido implementado utilizando patrones de componentes basados en la infraestructura de ICARO-T [24,8]. En cada componente autogestionado se encuentran cuatro tipos de agentes:

- **Manager:** Es el agente responsable de crear, terminar y gestionar el resto de agentes y recursos del componente "Management" descrito anteriormente en la sección 3.1.
- **Installer:** Tiene la responsabilidad de verificar las dependencias internas y de reparar los posibles errores que puedan ocurrir en la primera ejecución del componente auto-gestionado.
- **Runtime:** Es el agente responsable de verificar el correcto funcionamiento de las dependencias externas en tiempo de ejecución, delegando el control a un agente Repair cuando se detecta algún error.
- **Repair:** Es el agente que se responsabiliza de corregir los errores enviados por el agente Runtime. Por cada incidencia detectada, se crea un agente de este tipo.

Estos agentes utilizan una serie de recursos para llevar a cabo las distintas tareas de monitorización, análisis, etc. Además, cuando se detecta que hay que realizar alguna acción, p.e. reparación, se crean dinámicamente planes que determinan el tipo de medidas a tomar. A continuación se enumerarán los distintos tipos de recursos utilizados por los agentes. En la sección 3.6 se describe con más detalle la planificación.

3.5. Recursos

En cada componente auto-gestionado hay un conjunto de recursos que facilitan a los agentes la realización de sus diferentes tareas.

- **Recursos de monitorización:** Se encargan de obtener información en tiempo de ejecución de los componentes que suplen las dependencias.
- **Recursos de notificación:** Se encargan de gestionar las suscripciones con el resto de los componentes, así como de publicar y notificar el estado del componente.
- **Servicio de Nombrado:** Mantiene vínculos del tipo <nombre,entidad> para poder acceder uniformemente a las distintas entidades desde cualquier otra entidad, p.e. obtener la interfaz de un recurso desde un agente.
- **Analizadores de información de monitorización:** infieren, mediante casuística, el estado operacional de los componentes de los que el componente gestionado depende en ese momento, a partir de los informes de

monitorización.

- **Parsers XML:** Crean objetos computacionales que representan las dependencias descritas en los ficheros de descripción de dependencias internas y externas (IDDF, Internal Dependence Description File y EDDF, External Dependence Description File, respectivamente).
- **Resolutor de Web Services concretos:** Vincula dinámicamente con componentes Web Services que suplen las dependencias externas de servicios de aplicación.

3.6. Planificación

Los agentes crean planes dinámicamente que contienen las tareas a llevar a cabo cuando se detecta que puede ocurrir un funcionamiento distinto al esperado. Por ejemplo, cuando un componente detecta mediante monitorización que un componente del cual depende puede no estar disponible, se crea un plan que consiste en buscar un componente alternativo y establecer un vínculo con el mismo.

La Ilustración 8 describe el modelo de esta infraestructura referente a la planificación. Un *plan* es una secuencia de tareas. Una *tarea* se define como un operador que realiza algún cambio en el estado del entorno para conseguir un determinado objetivo.

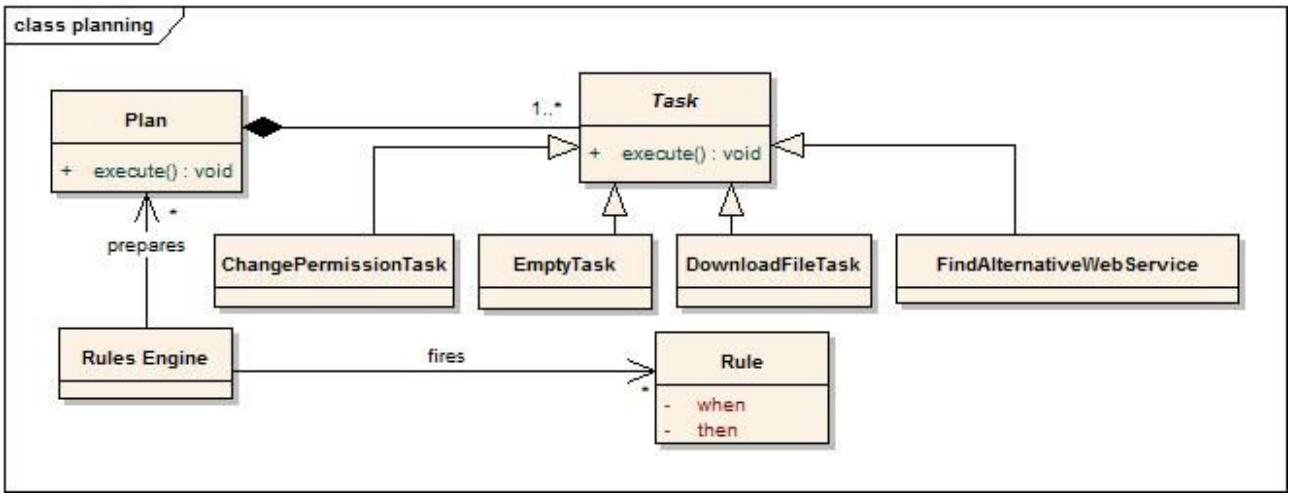

Ilustración 8: El modelo de planificación

La preparación de un plan consiste en encadenar diferentes tareas en secuencia. Este proceso se realiza dinámicamente por un agente cada vez que detecta alguna incidencia, analizando los informes de monitorización generados por los recursos de monitorización, con el fin de resolver dicha incidencia.

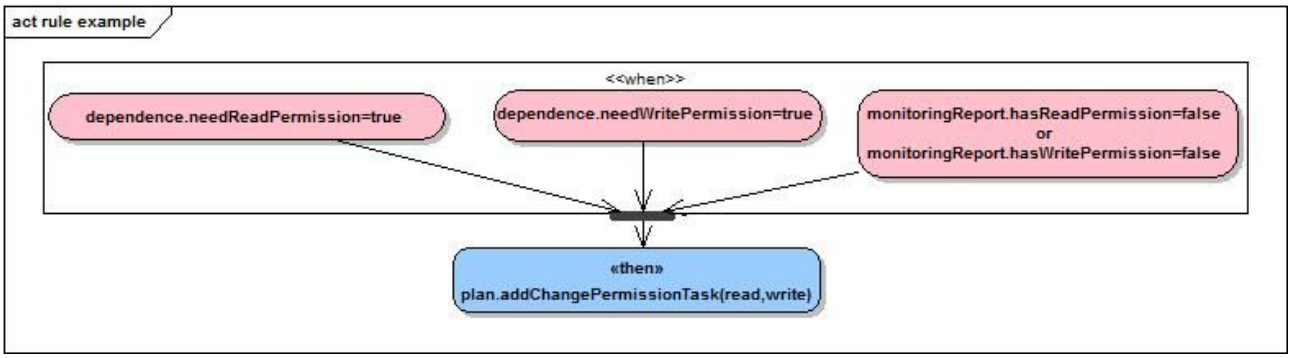

Ilustración 9: Un ejemplo de regla

Para saber qué tareas encadenar en cada situación, se ejecutan reglas del tipo “condición-acción”, en donde la parte izquierda de la regla contiene los posibles síntomas que puedan detectarse, y en la parte derecha la tarea a incluir en el plan que se está creando. Estas reglas se definen en un fichero de texto y se disparan con un motor de reglas basado en

el algoritmo RETE [5]. En esta implementación, se ha utilizado el motor Drools[13] y las reglas están definidas en el lenguaje Drools[13].

En la Ilustración 9 se muestra un ejemplo de regla. Se trata de la reparación a llevar a cabo cuando se detecta que un fichero no tiene los permisos de lectura/escritura requeridos por el componente. La tarea que se añade al plan consiste en corregir los permisos del fichero.

El conjunto de reglas y de tareas predefinidas puede ser fácilmente extendido para así modificar el comportamiento de los agentes ante distintas incidencias.

La preparación de un plan termina cuando no hay más reglas que disparar. Entonces el plan está listo para ser ejecutado, normalmente por el agente que lo crea.

4. Descripción de la Infraestructura

La Infraestructura IADESS agrupa los componentes que se utilizan para convertir un componente en auto-gestionado. Estos componentes, que aparecen en la Ilustración 10, son:

- **Management:** Se compone de agentes, recursos y un modelo, los cuales se describen en las secciones 3.4, 3.5 y 3.6 respectivamente y que se detallan más a fondo en el resto de secciones del capítulo.
- **ProxyOfControl:** controla el acceso al componente gestionado, evitando posibles malos usos en estados inapropiados, y provee información sobre el estado del componente auto-gestionado capturando las posibles excepciones técnicas que se pudieran lanzar.
- **Component:** es el componente original que se desea transformar.

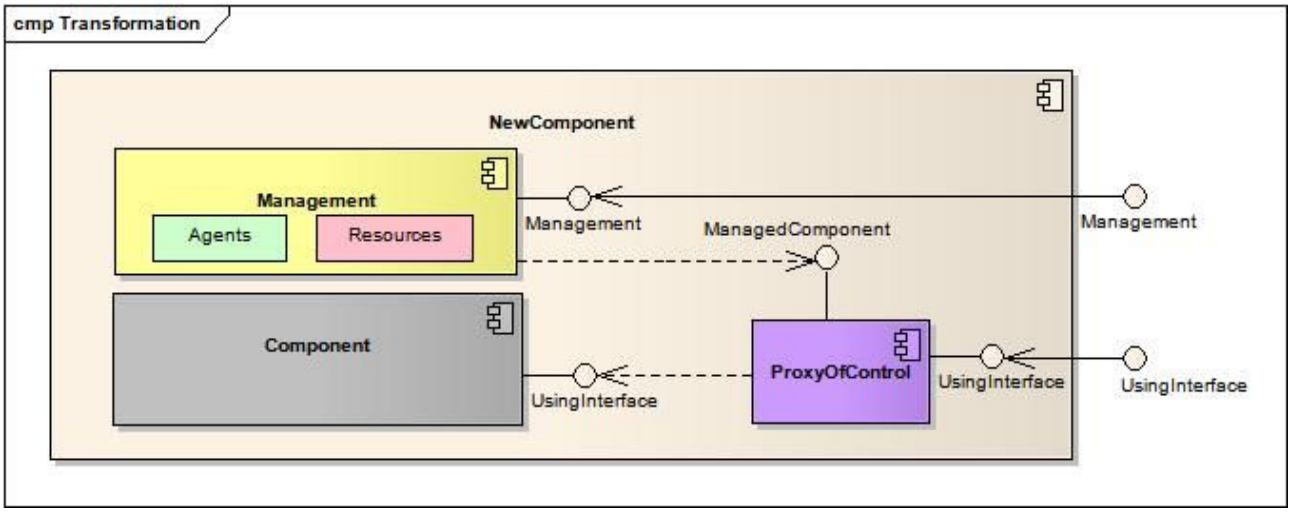

Ilustración 10: Arquitectura de un componente auto-gestionado

4.1. Descripción de la Arquitectura

La arquitectura del sistema propuesto se describe a continuación mediante cuatro puntos de vistas de acuerdo a la completitud necesaria de dicha descripción [10].

- **La Estructura:** Describe cómo está organizado el software por paquetes.
- **La Lógica:** Describe la lógica del sistema en diagramas de clases, es decir, entidades, agentes, recursos y objetos de transferencia de información.
- **La Dinámica:** Describe cómo interaccionan los agentes y recursos entre sí. También la comunicación que hay con otros sistemas de auto-gestión de otros componentes.
- **La Integración:** Describe la información necesaria (interfaces, configuraciones, etc.) para transformar un sistema existente en auto-gestionado usando la Infraestructura.

4.1.1. La Estructura

En la Ilustración 11 se muestra la estructura de paquetes de la infraestructura desarrollado y sus dependencias significativas. Como se puede observar, las dependencias son coherentes. El paquetes “agents” depende de la infraestructura ICARO-T ya que esta basada en ella y también de “Drools” ya que usa el motor como un recurso. Los paquetes “document” y “resources” dependen del paquete “javax::ws” ya que son estos los que implementan todo lo relacionado con Web Services, es decir, los agentes usan los recursos para comunicarse vía Red y no directamente.

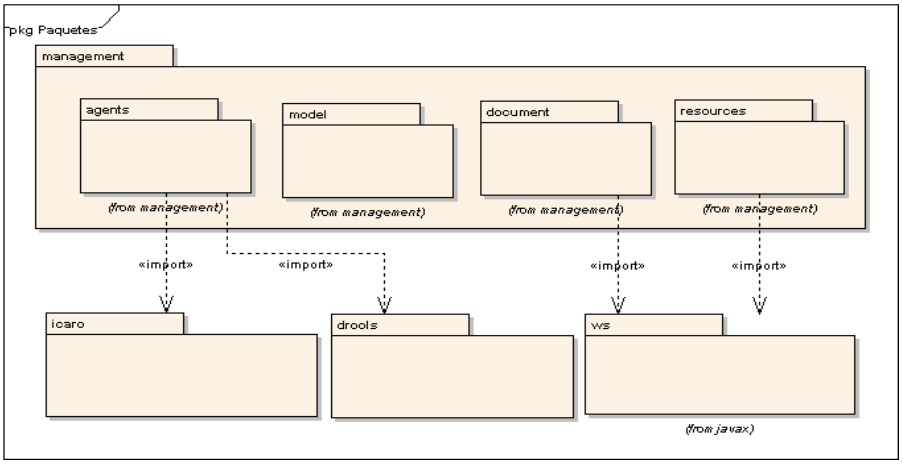

Ilustración 11: Estructura de Paquetes de la Infraestructura

Obsérvese que si se quieren cambiar alguna implementación o tecnología las dependencias en la arquitectura están bien definidas y los elementos afectos bien determinados.

4.1.2. La Lógica

A continuación se describen mediante diagramas de Clase UML la lógica de la infraestructura.

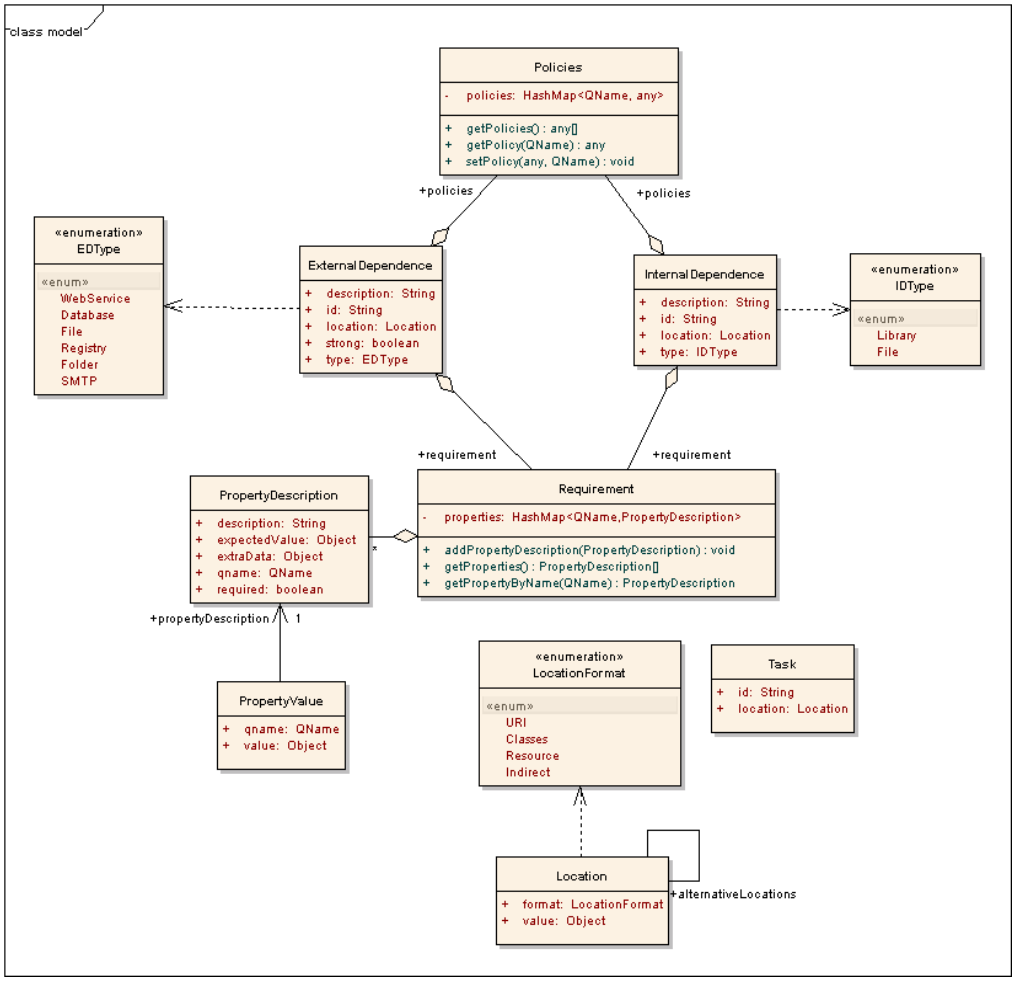

Ilustración 12: Diagrama del Modelo de Clases del Dominio

Las clases de diseño de la Ilustración 12 e Ilustración 13 corresponden aproximadamente con los conceptos manejados en el análisis realizado en el capítulo 3.

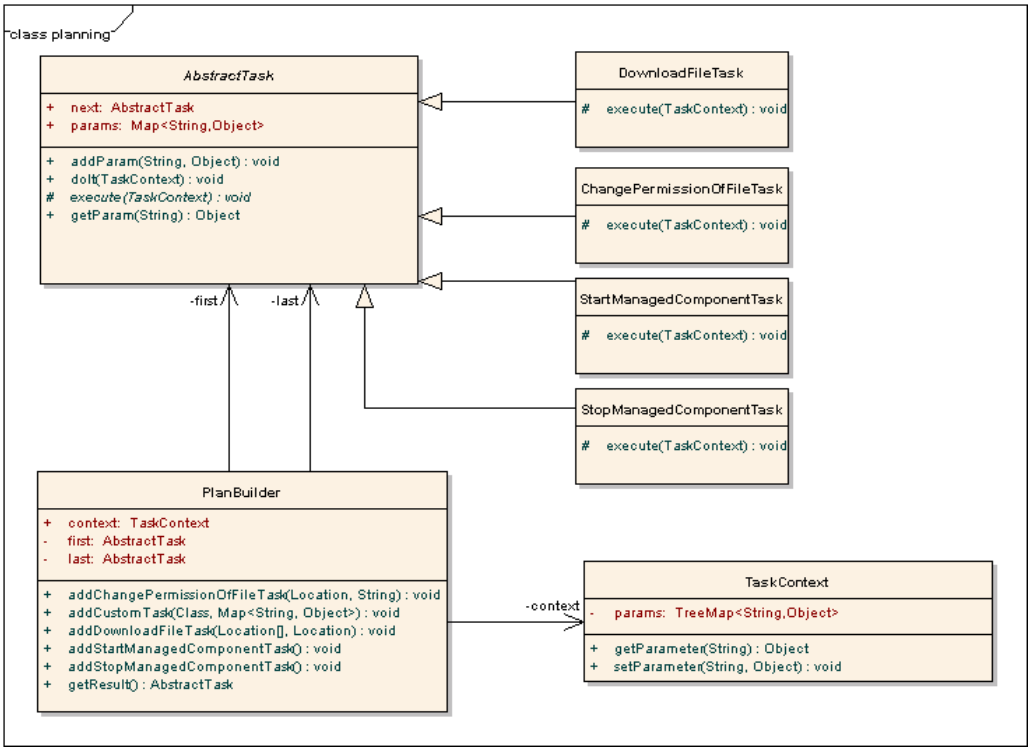

Ilustración 13: Modelo de Clases de Planificación

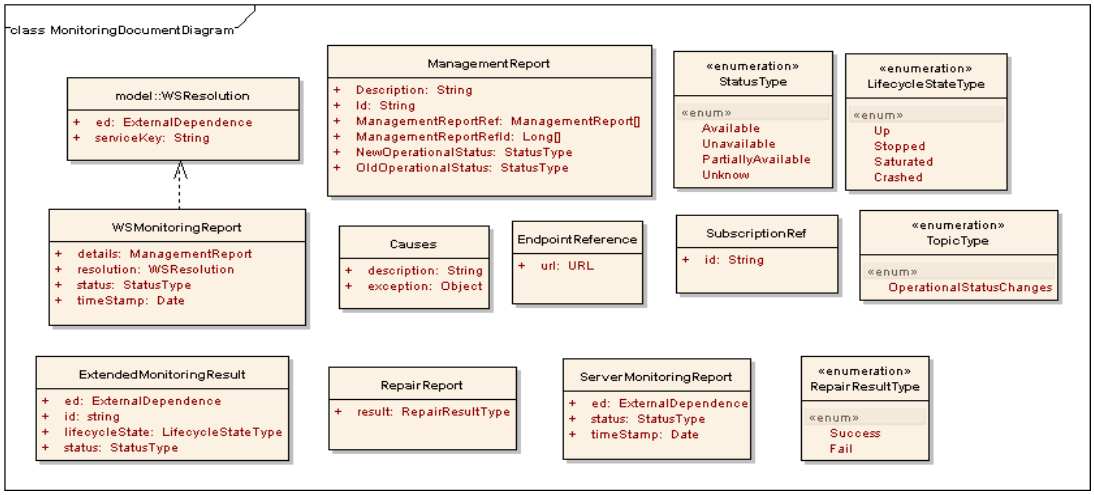

Ilustración 14: Estructuras relacionadas con la monitorización y otras informaciones

En la Ilustración 14 se describen clases de apoyo que son simplemente estructuras de datos para el intercambio de información a través de interfaces de recursos.

Los Estados Operacionales y del Ciclo de Vida de los Web Services se describen en las Tablas 1 y 2. Estas descripciones son tomadas de [18] y [27] respectivamente.

Los Estados Operacionales son aplicables a todos los componentes, ya sean Web Services o no, pero los Estados del Ciclo de Vida solo son aplicables a Web Services.

Estado Operacional	Descripción
Available	Este valor indica que el componente esta operativo con cualquier configuración, y listo para realizar todas sus tareas funcionales.
PartiallyAvailable	Este valor indica que el componente esta operativo parcialmente, es decir, que no esta listo para realizar todas sus tareas funcionales sino que solo un subconjunto de ellas.
Unavailable	Este valor indica que el componente no esta operativo, es decir, que no puede realizar ninguna tarea funcional. Puede que este parado o en fallo.
Unknown	Este valor indica que no ha sido posible conocer el estado actual del componente.

Tabla 1: Estado Operacional de un Componente[18]

Estado del Ciclo de Vida de Web Services	Descripción
Up	El agente proveedor del servicio esta listo para procesar cualquier invocación.
Stopped	El servicio ha sido intencionalmente parado (p.e. por motivos de administración).
Saturated	El agente proveedor del servicio tiene todos sus recursos ocupados y no puede aceptar nuevas invocaciones.
Crashed	El servicio no esta disponible debido a errores internos.

Tabla 2: Estado del Ciclo de Vida de un Web Services [27]

Los Recursos son creados a partir del “ResourceFactory” de la Ilustración 15.

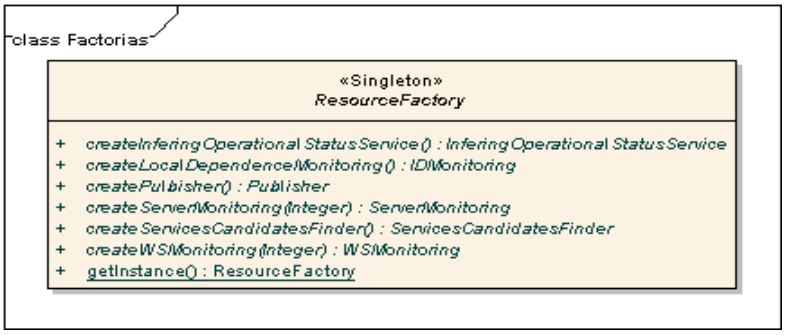

Ilustración 15: Factoría de Recursos

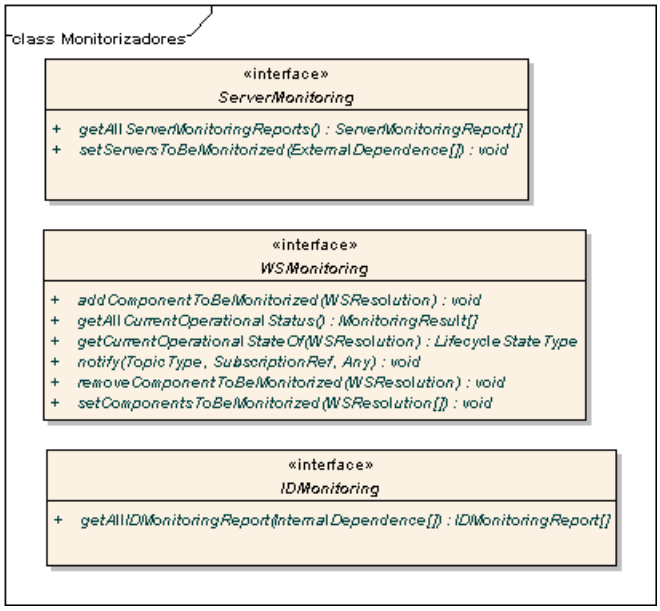

Ilustración 16: Recursos Monitores

Los Recursos Monitores "ServerMonitoring", "WSMonitoring" y "IDMonitoring" monitorizan los componentes que suplen las dependencias. Las interfaces mediante las cuales los agentes los usan se describe en la Ilustración 16.

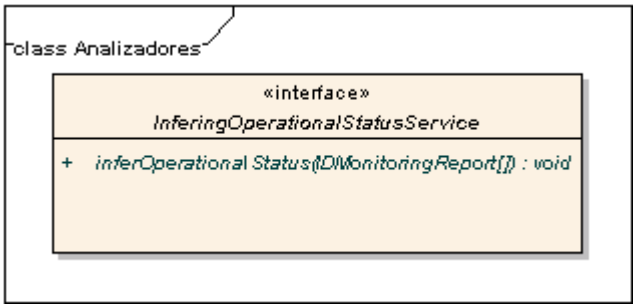

Ilustración 17: Recursos Analizadores

El recursos "InferingOperationalStatusService" se encarga de inferir el estado operacional de los componentes internos como ficheros, librerías, etc... a partir del estado de su propiedades. En la Ilustración 17 se muestra la interfaz de uso.

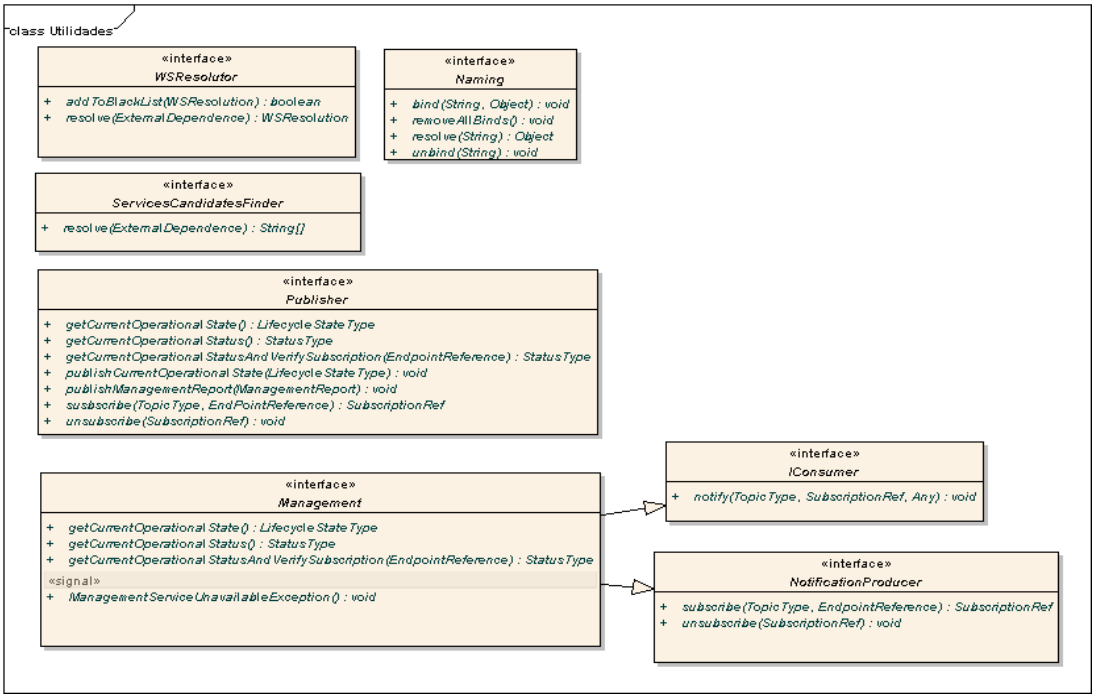

Ilustración 18: Recursos de Apoyo

En la Ilustración 18 se muestran las interfaces de uso de una serie de recursos de apoyo que utilizan los agentes para comunicarse vía Web Services, para gestionar el vinculo dinámico y para el nombrado de recursos y agentes en el nodo.

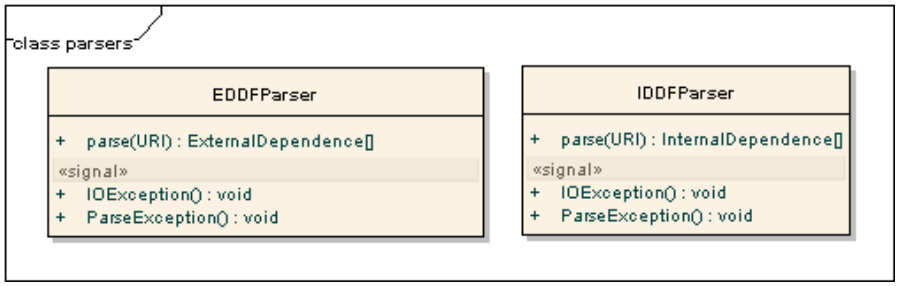

Ilustración 19: Recursos Parsers de XMLs

Y finalmente en la Ilustración 19 se muestran los recursos encargados de crear clases del modelo a partir de ficheros XML con información al respecto.

4.1.3. La Dinámica

A continuación se describirá el ciclo de funcionamiento normal de los agentes de la infraestructura:

1. El agente Manager registra los recursos en el servicio de nombrado.
2. El agente Manager crea y registra los agentes Installer y Runtime.
3. El agente Manager arranca el agente Installer (Ilustración 20)

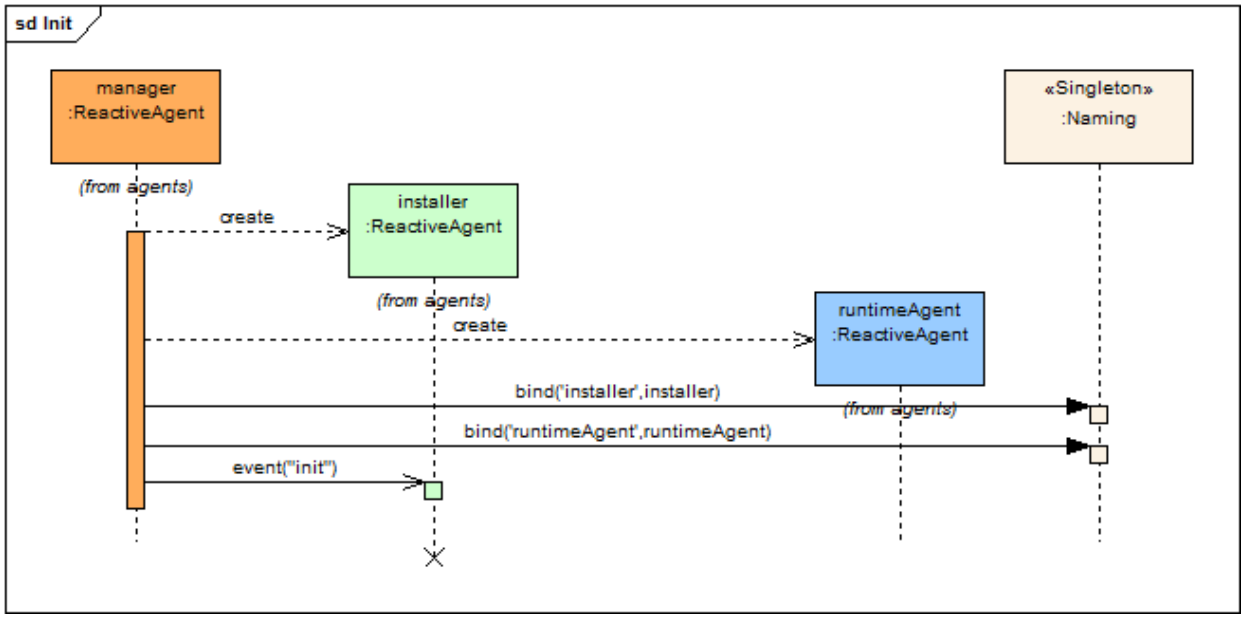

Ilustración 20: Arranque del Installer Agent

4. El agente Installer verifica las dependencias internas y repara los posibles errores que puedan ocurrir. En caso de error crítico, reporta al Manager, el cual se encarga de terminar la parte de auto-gestión del componente.
5. El agente Installer realiza ciclos de monitorización, análisis y reparación de dependencias internas hasta llegar a una situación estable.
6. Cuando el Installer termina sus tareas, avisa al Manager. (Ilustración 21)
7. El agente Manager termina al agente Installer y crea, registra y arranca al agente Runtime.

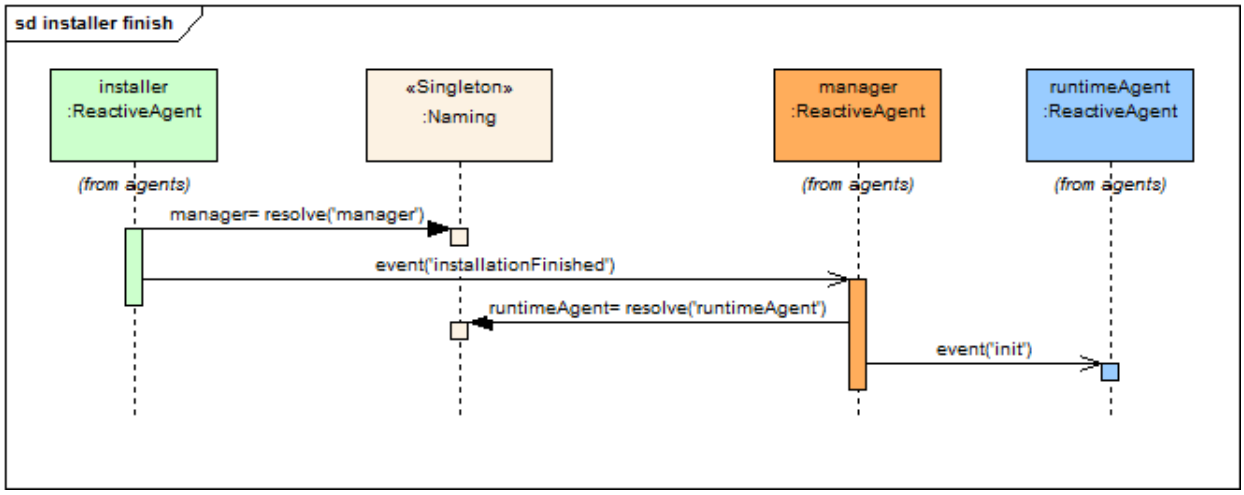

Ilustración 21: Terminación del Installer Agent y Arranque del Runtime Agent

8. El agente Manager crea, registra y arranca a un agente Repair.
9. El agente Runtime envía la información de reparación al agente Repair.(Ilustración 22)

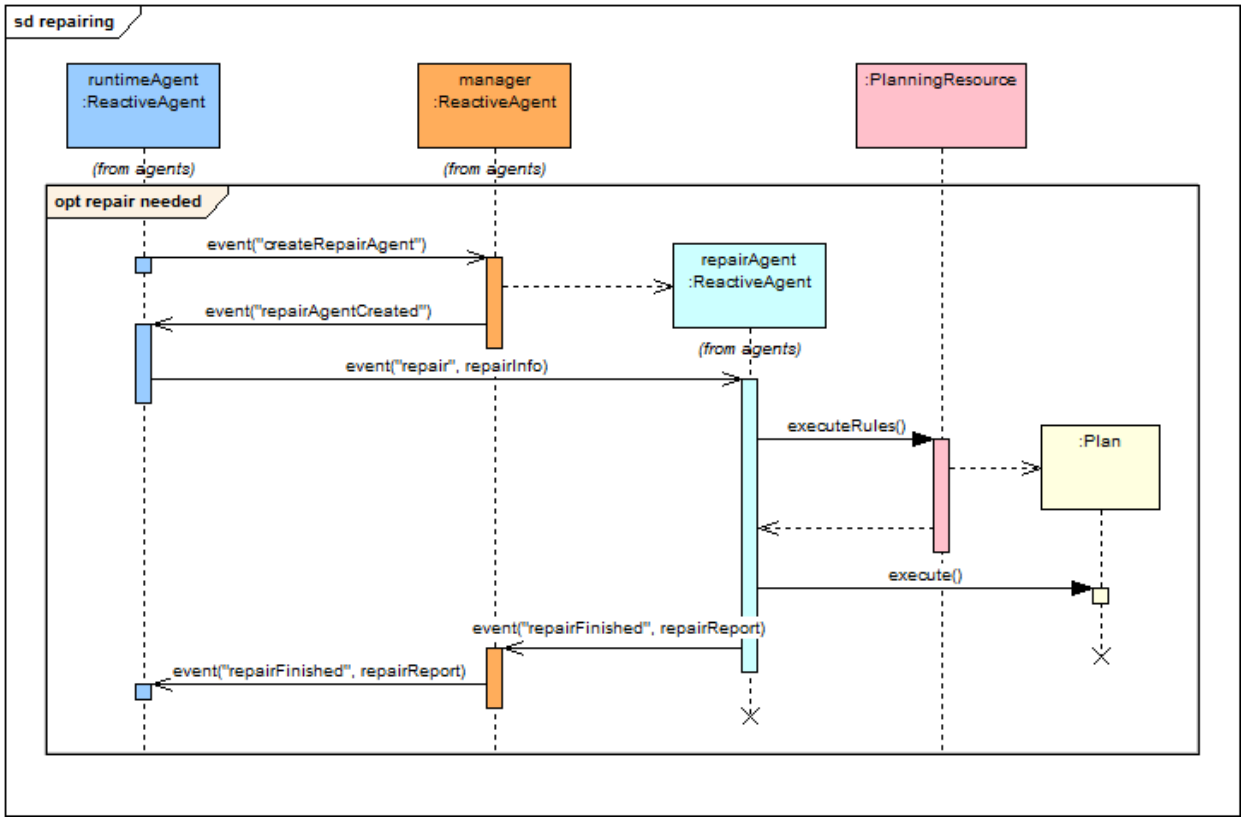

Ilustración 22: Creación de un Repair Agent en caso de reparación

10. El agente Repair intenta reparar la incidencia, envía un informe al agente Runtime y avisa al agente Manager.
11. El agente Manager termina al agente Repair.
12. El agente Runtime realiza un nuevo ciclo de monitorización y analiza el informe enviado por el Runtime.

En la Ilustración 23 se describe a alto nivel el funcionamiento del Runtime Agent, resaltando los eventos que envía al resto de agentes y las fases que realiza. En la fase de monitorización, recoge pide información a los recursos de monitorización de dependencias externas (*WSMonitoring* y *ServerMonitoring*). En la fase de análisis, analiza esta información, junto a informes de reparación anteriores e información de su estado operacional actual con el fin de detectar posibles mal funcionamientos del componente. Por otro lado, si detecta un cambio en el estado, éste se publica para que la información actualizada esté disponible para el resto de componentes auto-gestionados.

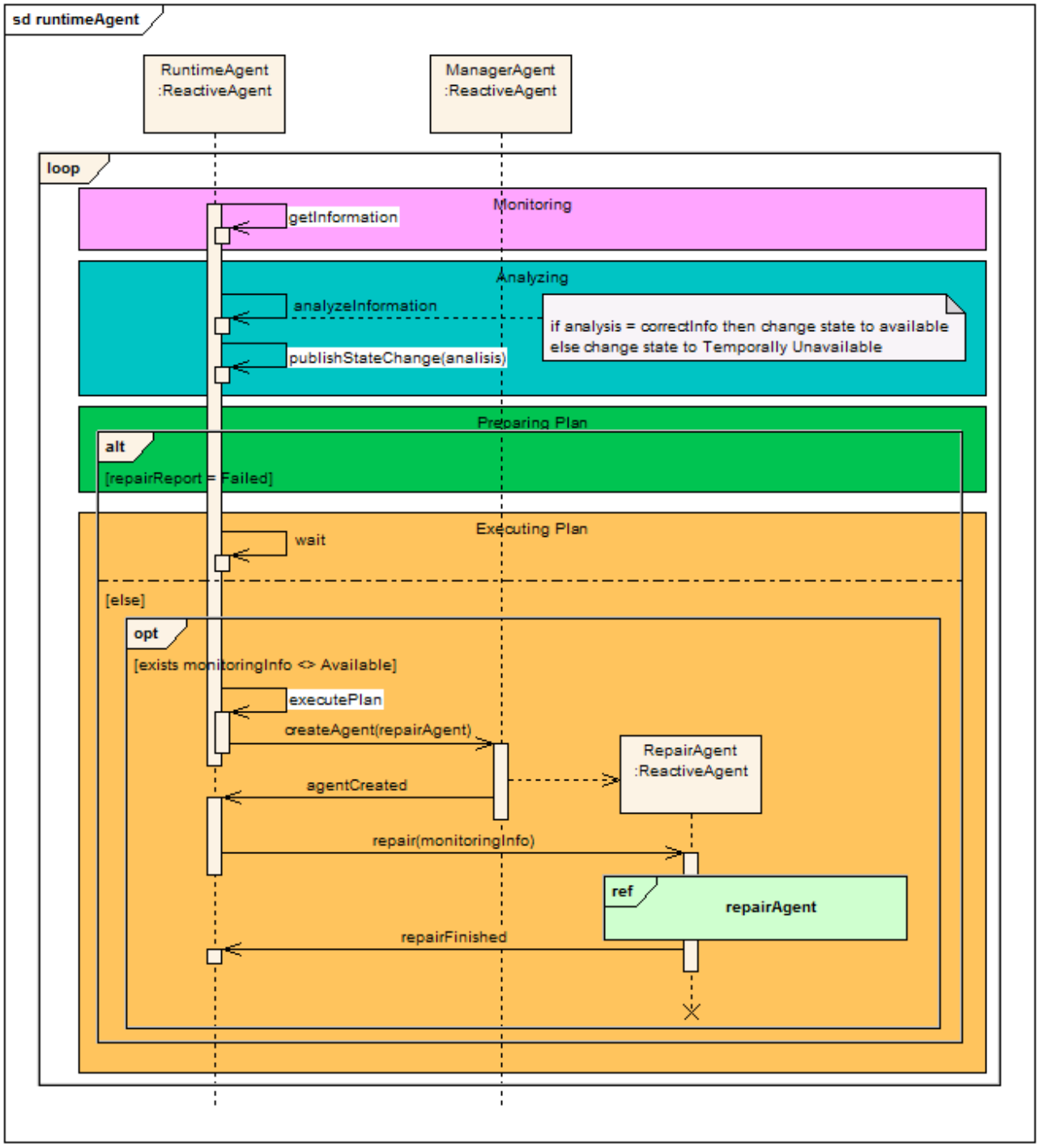

Ilustración 23: Ciclo de funcionamiento del Runtime Agent

En la fase de preparación, se decide si es necesario crear un agente Repair para intentar reparar una incidencia. Si se detecta que un intento de reparación anterior no ha solucionado la incidencia, entonces se espera un tiempo y se vuelve a empezar el ciclo.

La Ilustración 24 muestra las fases que realiza un agente Repair. Las fases de monitorización y análisis se encargan de complementar la información enviada por el agente Runtime. En la fase de preparación del plan se crea el plan tal como se describe en la sección 3.6. Al terminar la ejecución de las reglas, se obtiene el plan que se ejecuta en la siguiente fase. Por último, el agente Repair envía un informe de reparación al agente Runtime, indicando si el plan se ha ejecutado con éxito o no.

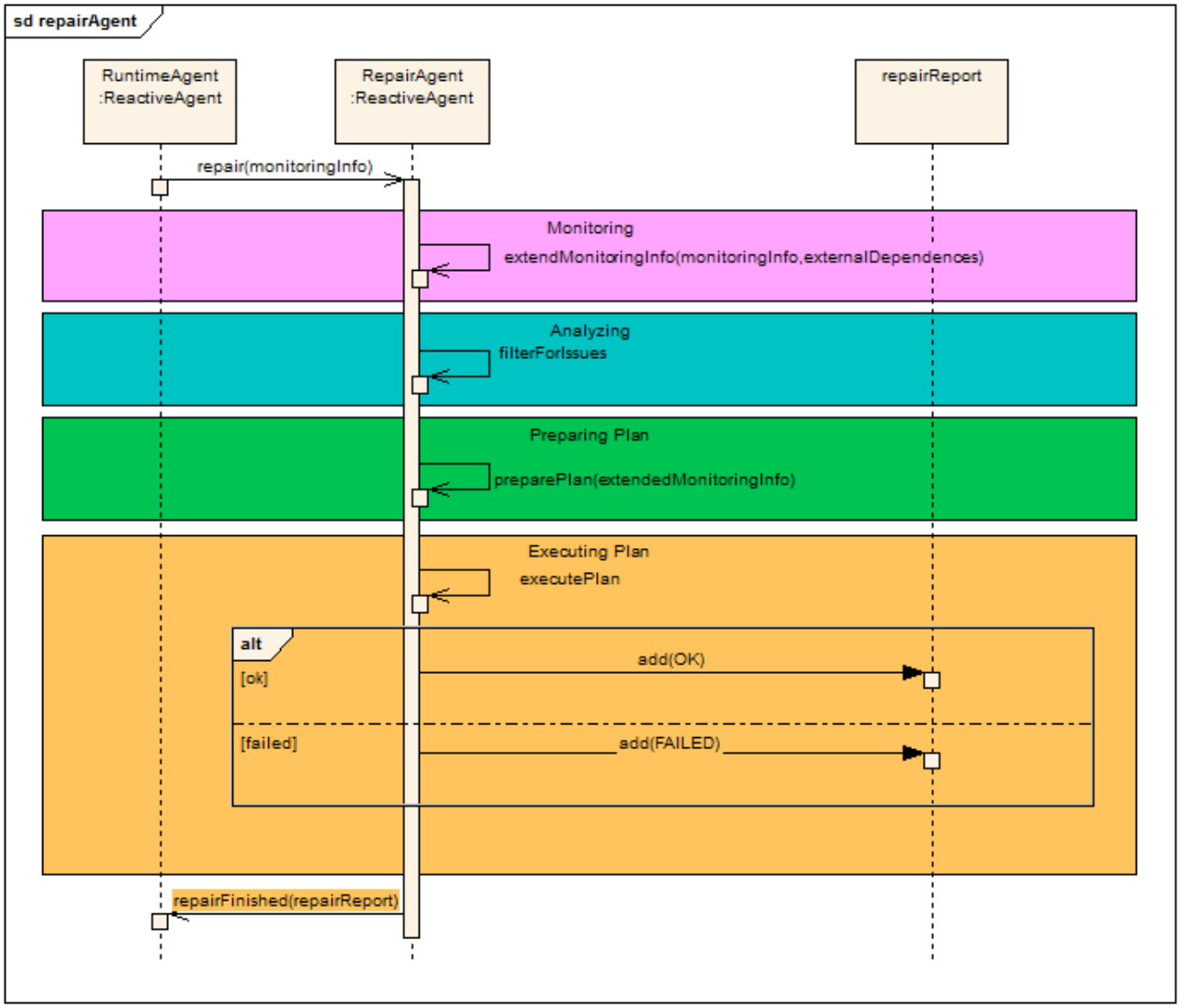

Ilustración 24: Ciclo de funcionamiento del Repair Agent

A continuación se desribirán las máquinas de estados de los agentes Manager (Ilustración 25), Installer (Ilustración 26), Runtime (Ilustración 27) y Repair (Ilustración 28).

El primero de ellos, el Manager, implementa un ciclo en el que inicializa los recursos y los agentes Installer y Runtime al inicio de la aplicación, y a partir de ahí comienza a delegar las tareas de gestión en los distintos agentes. En un principio delega en el Installer, para que verifique el funcionamiento de las dependencias internas, y cuando éste termina, arranca el Runtime, el cual se encarga de monitorizar el funcionamiento de las dependencias externas. En ese momento, el agente queda detenido en espera de solicitudes para crear nuevos agentes Repair y avisar de la finalización de su tarea de nuevo al Runtime. El agente Manager sirve como coordinador para que las tareas se realicen en su orden adecuado y es el encargado de crear al resto de agentes.

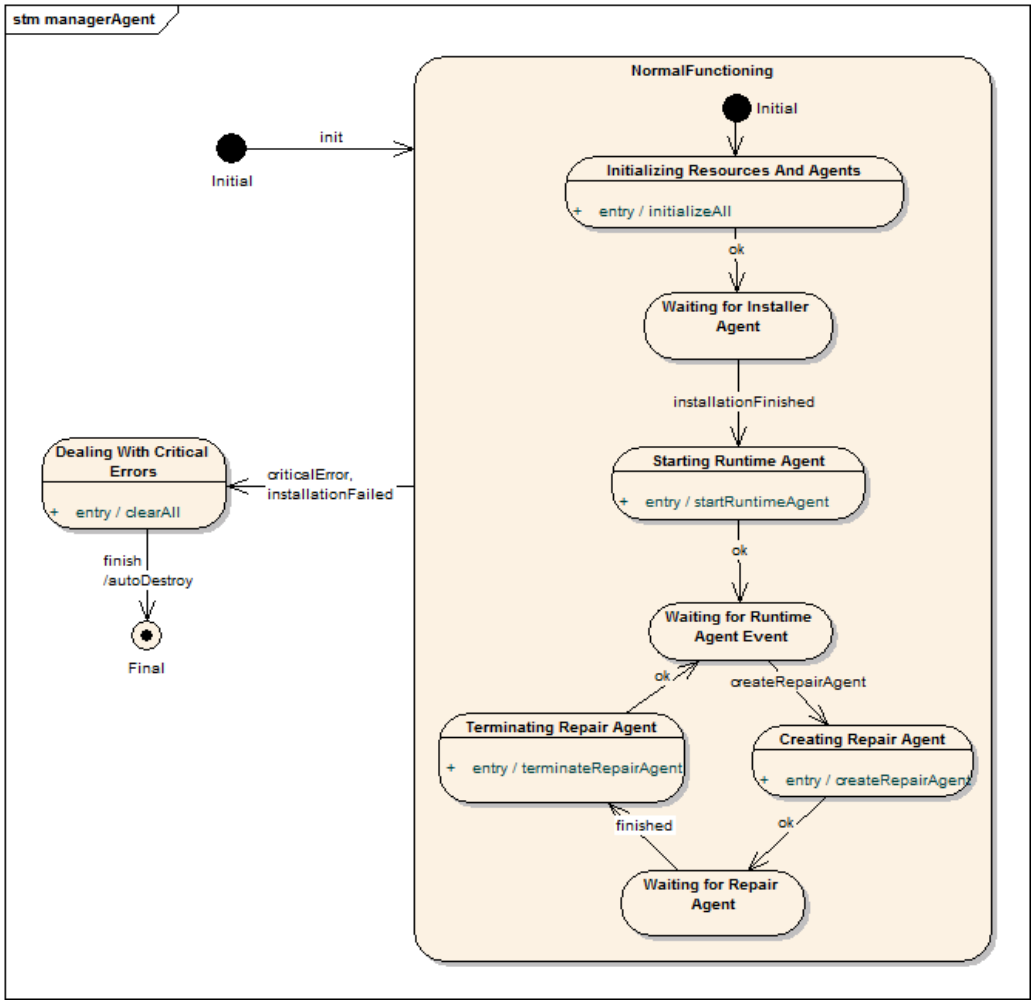

Ilustración 25: Máquina de estados del Manager Agent

El Installer, comienza cargando el fichero de dependencias internas XML para conocer qué es lo que debe monitorizar. Luego toma informes de estado de cada uno de los objetos de los que depende, con los que comprueba si estos están en el estado adecuado para el funcionamiento del componente. Si no es así, planifica la secuencia de tareas necesaria para intentar alcanzar el estado deseado y vuelve de nuevo a comprobar. Si tras una serie de ciclos no se consigue alcanzar el estado deseado, entonces informa de la situación de fallo y finaliza. Si ha conseguido solventar el problema, informa del éxito y finaliza igualmente.

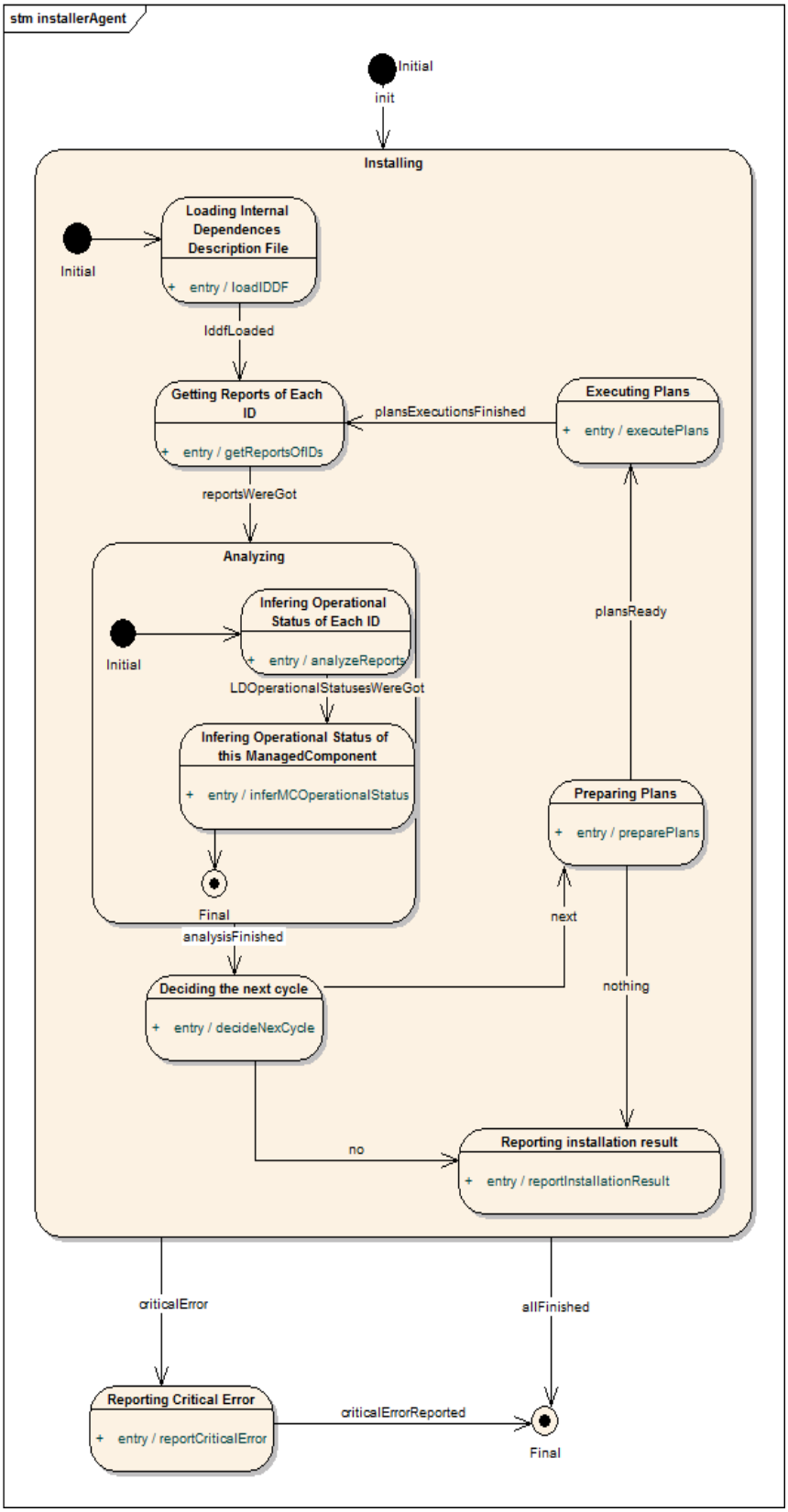

Ilustración 26: Máquina de estados del Installer Agent

El Runtime comienza consultando el status del componente y luego carga el fichero de dependencias externas XML. Luego, intenta inicializar los recursos de monitorización con aquellos objetos encontrados que soporten las

dependencias requeridas. Si no existen dichos objetos, se esperará en estado de No Disponibilidad hasta que existan.

En ese momento entra en un ciclo en el que pregunta el estado de disponibilidad de los objetos de los que depende a los recursos de monitorización. Si encuentra alguna información interesante (cambios de estado u otros eventos) pasa al estado de determinar estado del componente. En este estado halla el estado del componente en función del de los objetos de los que depende.

Si fallan objetos de los que sólo depende parcialmente, el estado del componente será parcialmente disponibilidad, pero si fallan objetos críticos para su funcionamiento, el estado será no disponible.

Sin embargo, antes se publica un estado intermedio de espera, en el que muestra que está intentando resolver el problema. La publicación avisa a los componentes suscritos del nuevo cambio de estado. Si el cambio de estado ha sido a un estado de espera para la reparación, entonces se requiere al Manager para crear un agente Repair, al que se le pasan los informes de error para que intente solucionarlos. Si la información recogida indica que un intento de reparación anterior ha fallado, entonces se pasa a estado de No Disponibilidad hasta transcurrido un cierto tiempo.

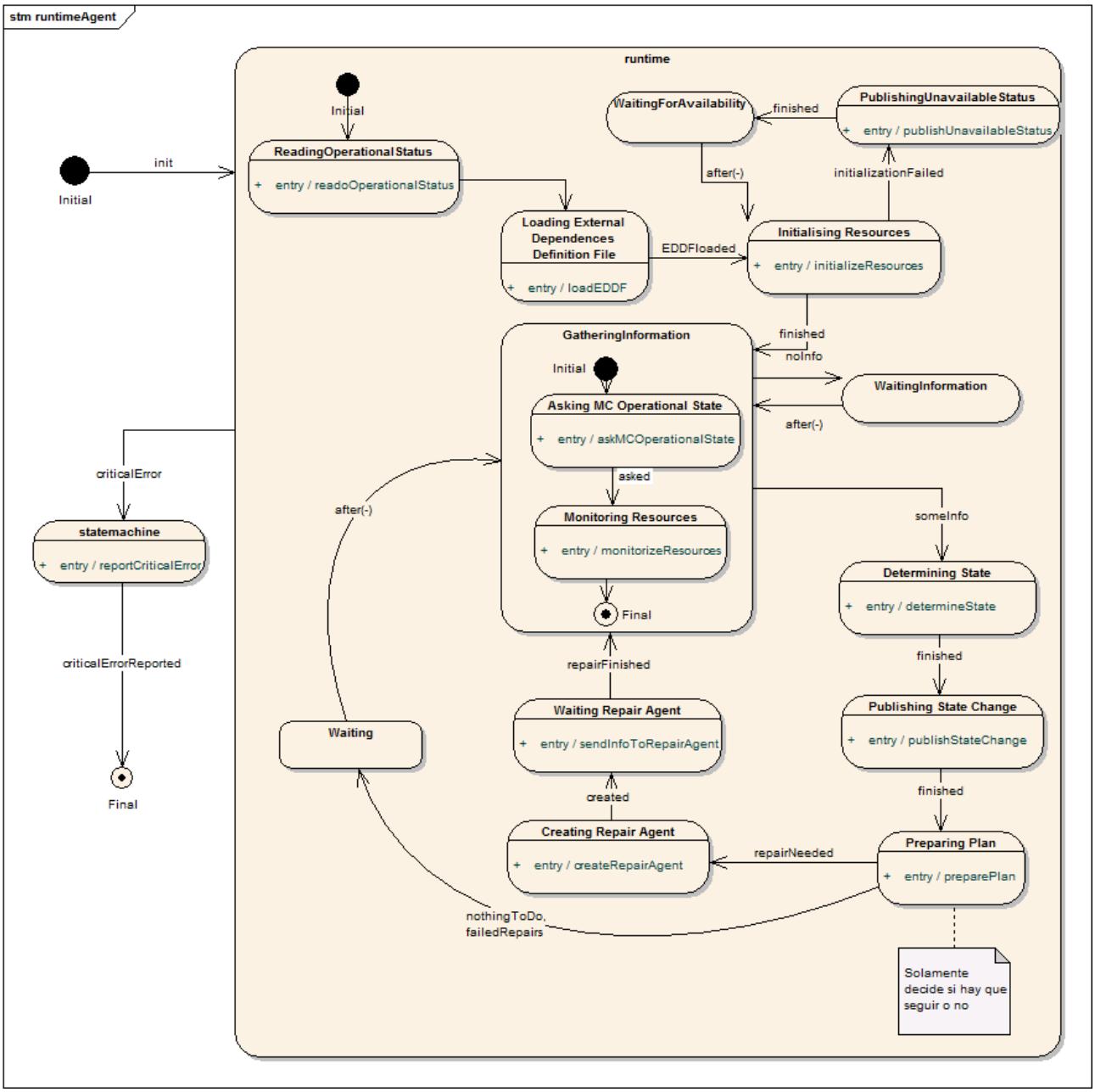

Ilustración 27: Máquina de estados del Runtime Agent

El agente Repair es el encargado de las reparaciones necesarias del componente. Para ello lee el estado operacional del componente y recoge la información de reparación del agente Runtime. A partir de ahí se dedica a hacer un ciclo más exhaustivo de monitorización y análisis para hallar las causas finales del error. Con ellas, utiliza las reglas para preparar un plan y posteriormente lo ejecuta. Tras la ejecución, envía de vuelta los resultados obtenidos. No intenta otras vías de reparación, le deja esta decisión al agente Runtime.

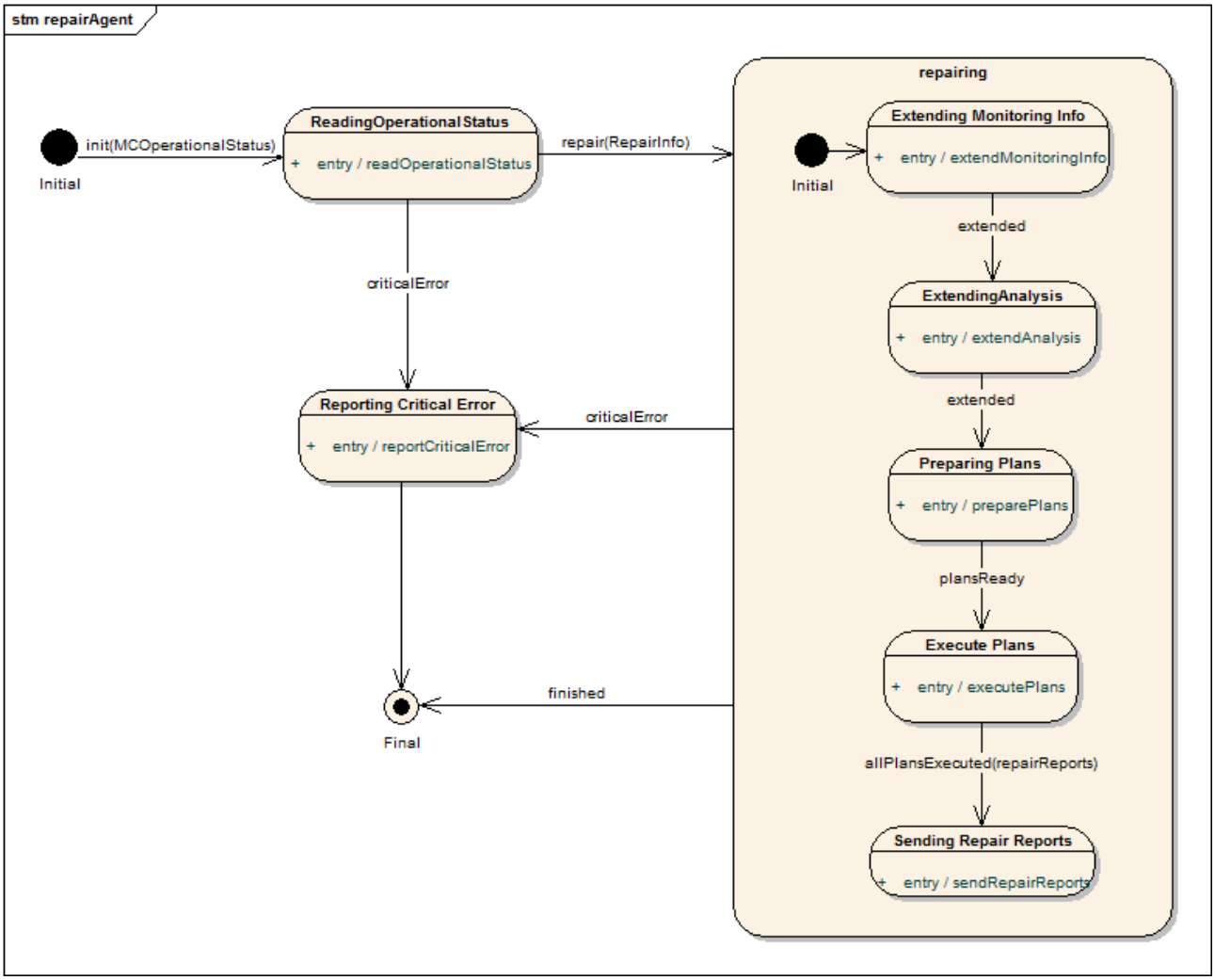

Ilustración 28: Máquina de estados del Repair Agent

4.1.4. La Integración

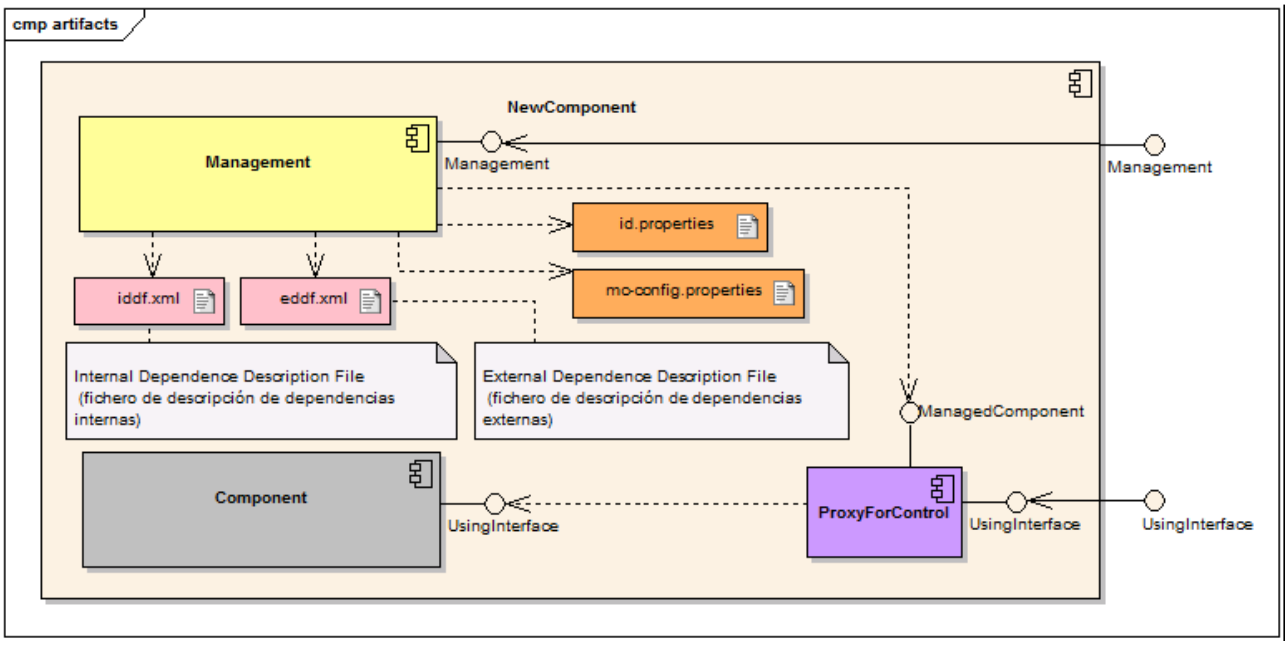

Ilustración 29: Artefactos en un componente auto-gestionado

La aplicación de la infraestructura a un componente con el fin de transformar un componente en auto-gestionado (Ilustración 29), consiste fundamentalmente en:

- proporcionar la interfaz *ManagedComponent* que permita gestionar el arranque, la terminación, obtener el estado operacional del componente gestionado y recoger información (por ejemplo, excepciones) sobre posibles incidencias ocurridas en el uso del componente. En los diagramas de las ilustraciones 34, 35 y 36, se muestra una recomendación de diseño para el componente *ProxyForControl*, que se encarga de proveer dicha funcionalidad.
- declarar las dependencias entre el componente gestionado y sus partes internas (dependencias internas). Esto se realiza proporcionando un fichero XML que instancia el esquema iddf.xsd descrito en la Ilustración 30.
- declarar las dependencias entre el componente gestionado y el resto de componentes del sistema (dependencias externas), proporcionando un fichero XML que instancie el esquema eddf.xsd.(ilustraciones 32 y 33)
- describir los valores esperados de un conjunto de propiedades de los mismos.

El resto de artefactos contienen información sobre el identificador con el que el componente auto-gestionado se registra en el directorio UDDI[17] (*id.properties*) e información de configuración que permite al desarrollador establecer parámetros como pueden ser frecuencia de monitorización, parámetros de tareas de los planes, etc.

Las propiedades que soporta actualmente la infraestructura desarrollada son las siguientes: CanRead, CanWrite, XMLSyntaxChecked y son aplicables sólo a ficheros. En el caso de XMLSyntaxChecked se puede especificar información adicional respecto a la localización del Schema del fichero a chequear.

La información en UDDI debe estar estructurada de la siguiente manera: los servicios deben tener un "BindingTemplate" para especificar la implementación de su interfaz de uso y un "BindingTemplate" para especificar la implementación de su interfaz de gestión si es que la tiene. La infraestructura siempre trata de usar la interfaz de gestión si la encuentra para sus actividades de monitorización, pero en caso de que no la encuentre trata de verificar el estado realizando conexiones HTTP sobre la interfaz de uso.

Las implementaciones de las interfaces de gestión deben hacer referencia a un tModel común que describe dicha interfaz, en el caso del prototipo se ha usado un tModel común con UUID “uuid:410F68F0-D804-11DC-A8F0-FB12590D29EB”.

Ilustración 30: XML Schema del fichero de descripción de dependencias internas

Una dependencia interna puede ser un fichero (*File*) o una librería (*Library*). Cada fichero se describe con un identificador, una descripción, una localización (*Location*) y una lista de propiedades.

La localización tiene como atributos valor (de tipo AnyURI) y formato de la misma. Se incluye como elemento una localización alternativa.

Las propiedades son:

- se requiere permiso de lectura
- se requiere permiso de escritura
- se requiere validación XML (se proporciona la localización del XML Schema)

Una librería tiene un identificador, una descripción y en la localización se describen las clases de dicha librería de las que se debe comprobar su disponibilidad.

En la Ilustración 31 se muestra un ejemplo de fichero iddf.xml para el servicio de gestión de usuarios del prototipo descrito en el Anexo A.

```
<iddf:InternalDependences xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:mc:iddf schemas/iddf.xsd"   xmlns:iddf="urn:mc:iddf"
                           xmlns:fp="urn:mc:iddf:file:properties:permission"
                           xmlns:fv="urn:mc:iddf:file:properties:checking"
                           xmlns:jcl="urn:mc:iddf:library:location:java:classes"
                           xmlns:task="urn:mc:iddf:dependence:policies:task">
    <iddf:File Id="persistenceConfig">
        <iddf:Description>Persistence Configuration</iddf:Description>
        <iddf:Location format="uri"
    value="file:///opt/tomcat/webapps/GestionUsuarios/WEB-INF/classes/META-INF/persistence.xml">
            <iddf:AlternativeLocations>
                <iddf:Location format="uri" value="http://localhost:8084/persistence.xml"/>
            </iddf:AlternativeLocations>
        </iddf:Location>
        <iddf:Policies>
        </iddf:Policies>
        <iddf:Properties>
            <fp:CanRead value="true"/>
            <fv:XMLSyntaxChecked value="true">
                <fv:Schema>
                    <iddf:Location format="uri" value="file:///home/iadess/config-schema.xsd"/>
                </fv:Schema>
            </fv:XMLSyntaxChecked>
        </iddf:Properties>
    </iddf:File>
    <iddf:Library Id="logginglib">
        <iddf:Description>Logging Library</iddf:Description>
        <iddf:Location format="classes">
            <jcl:Class name="org.apache.commons.logging.LogFactory"/>
            <jcl:Class name="org.apache.commons.logging.impl.SimpleLog"/>
        </iddf:Location>
    </iddf:Library>
</iddf:InternalDependences>
```

Ilustración 31: Ejemplo de fichero iddf.xml

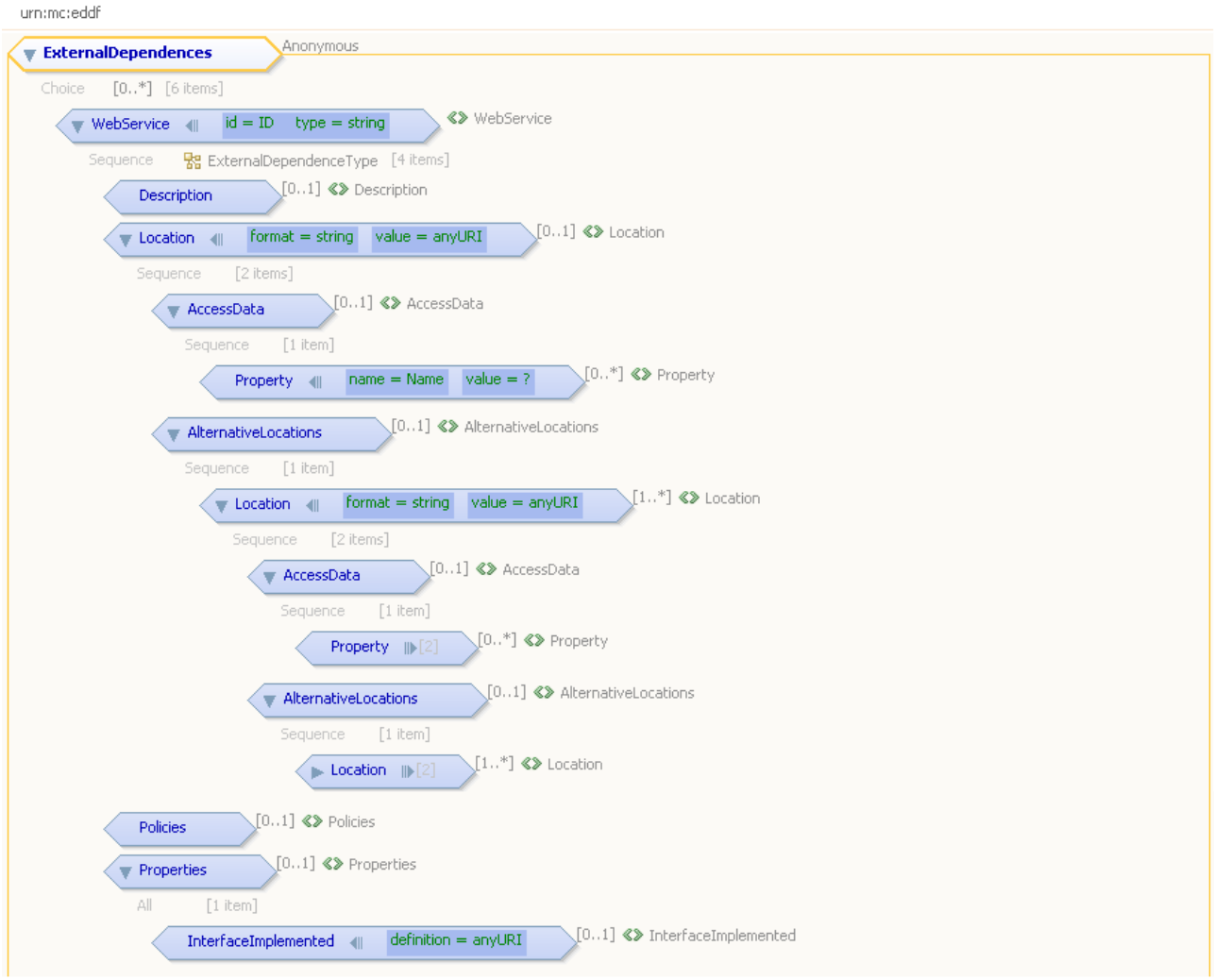

Ilustración 32: XML Schema del fichero de descripción de dependencias externas (I)

En las ilustraciones 32 y 33 se muestra el esquema del fichero de descripción de dependencias externas. En éste se indica las características que se necesita conocer de los componentes de los que depende. Estos son Web Services, base de datos (Database), ficheros, carpetas o servidores de correo SMTP. La información necesaria de cada componente consiste en un identificador, una descripción, información de localización (dónde se encuentra, cómo se accede), propiedades del componente, etc. Los componentes Web Services proporcionan además la interfaz WSDL implementada.

Ilustración 33: XML Schema del fichero de descripción de dependencias externas (II)

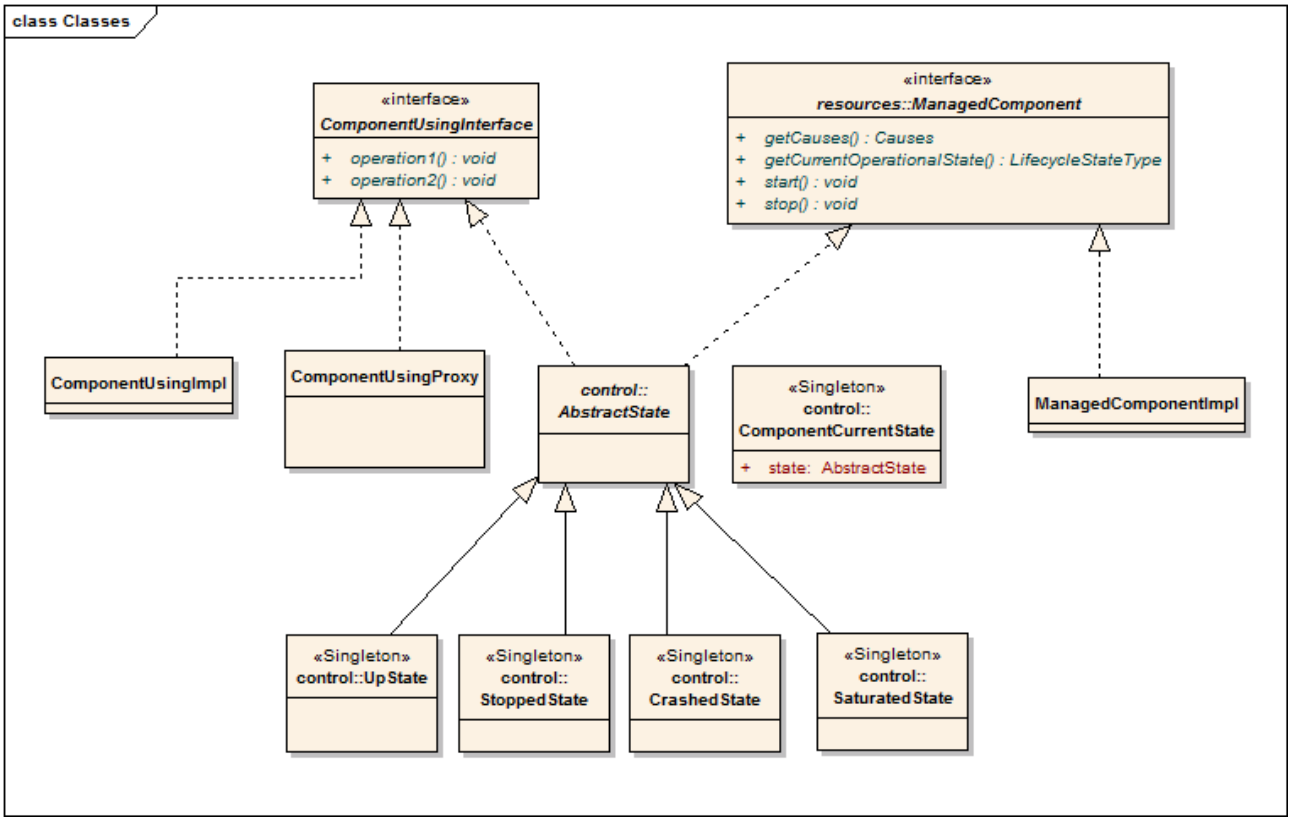

Ilustración 34: Diagrama de clases del componente ProxyForControl

ComponentUsingInterface es la interfaz de uso del componente gestionado. El componente *ProxyForControl* se encarga de proveer esta interfaz al resto de componentes del sistema. *ManagedComponent* es la interfaz que se proporciona al componente *Management* para operaciones de gestión sobre el componente gestionado. El control del componente *ProxyForControl* está diseñado según el patrón State[7] (Ilustración 34). Cada uno de los estados de dicho control corresponden con los descritos en la Tabla 2.

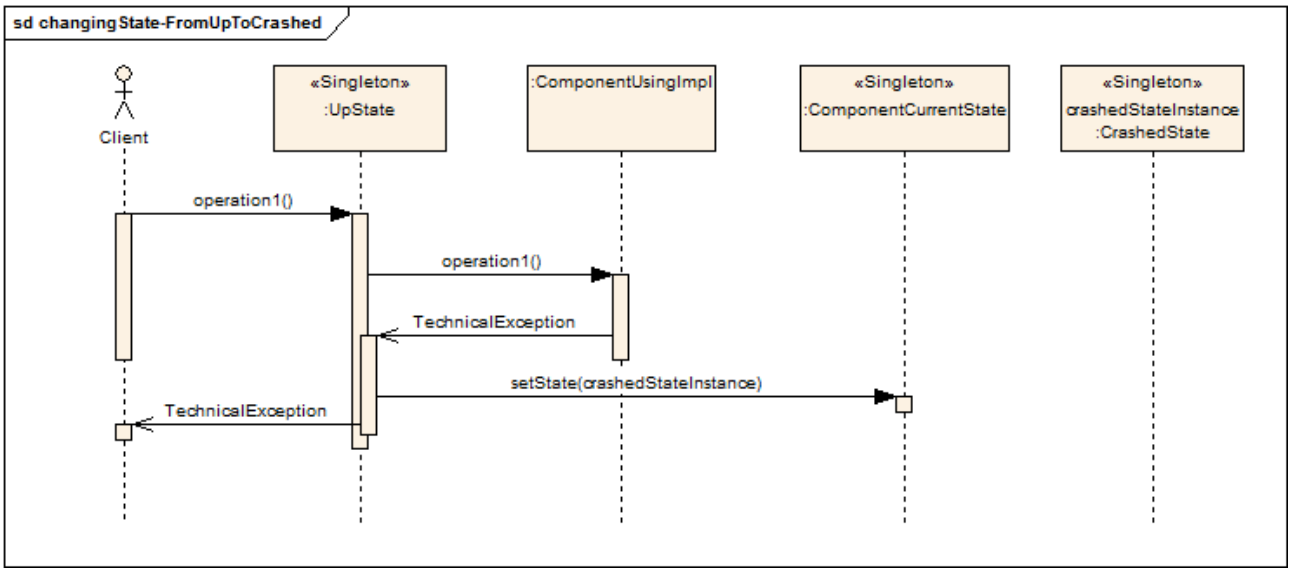

Ilustración 35: Cambio de estado al producirse un error en el uso del componente gestionado

Cuando un cliente accede a la interfaz de uso del componente auto-gestionado, el componente *ProxyForControl* gestiona el estado del componente gestionado. En la se muestra como se cambia de estado cuando se produce un error en el uso del componente gestionado (Ilustración 35).

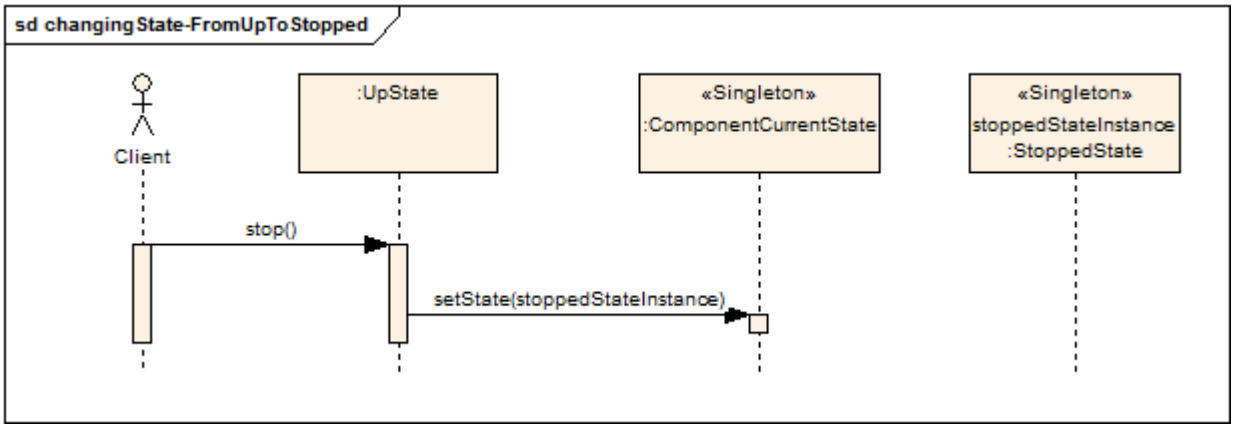

Ilustración 36: Cambio de estado al parar el componente gestionado

Cuando el componente *Management* pide que se pare el componente gestionado (Ilustración 36) -por ejemplo, esto puede producirse durante la ejecución de un plan como medida de reparación de una incidencia- el componente cambia el estado de *Up* (activo) a *Stopped* (parado).

4.2. Herramientas auxiliares de la infraestructura

Además de la infraestructura desarrollada, se ha desarrollado una Aplicación llamada “ConsoleAdmin” que tiene la siguiente funcionalidad:

- **Monitorización:** Monitoriza el estado de los componentes de un sistema distribuido, tanto a los que tienen la capacidad de auto-gestionarse como los que no.
- **Predicción:** Predice cual será el estado operacional de los componentes cuando se detecta un fallo en alguno de ellos.

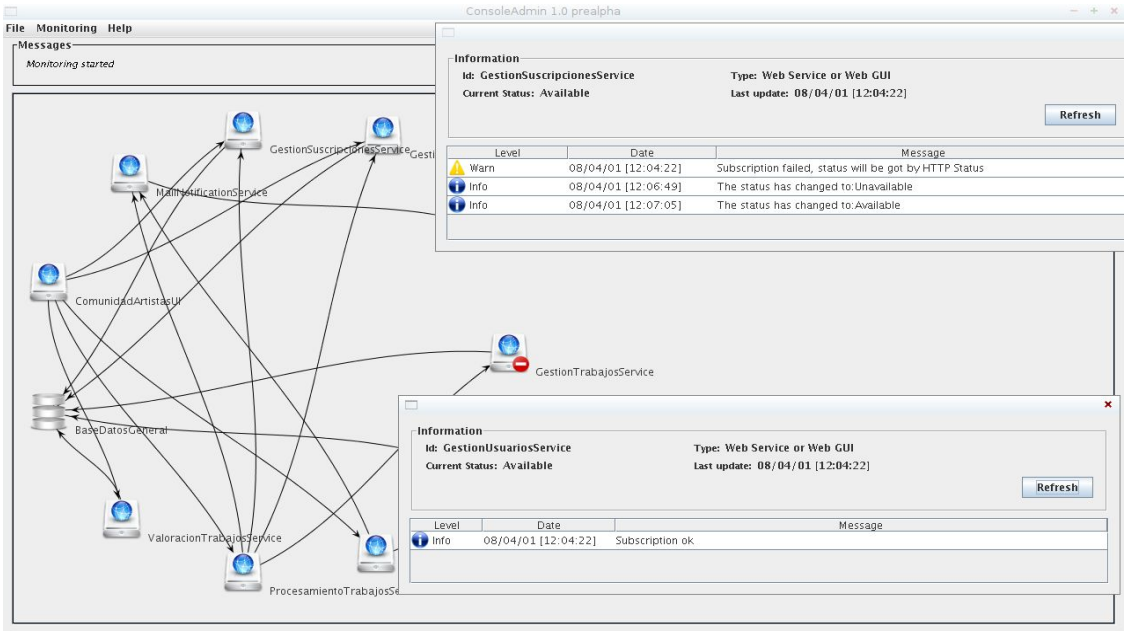

Ilustración 37: "ConsoleAdmin" en acción

El “ConsoleAdmin” para obtener la información utiliza los mismo métodos que la infraestructura utiliza para recopilar información del estado de las dependencias del componente a gestionar. Es decir, el “ConsoleAdmin” se comporta como

si “dependiera” de todos los componentes del sistema a monitorizar.

La estructura de datos utilizada, en la cual se basa esta herramienta, es el grafo. A partir de la información de los componentes y sus dependencias directas la herramienta construye un grafo que representa la red de interdependencias entre componentes en el sistema. En el grafo, los nodos representan componentes que se monitorizan y las conexiones representas posibles relaciones de uso entre estos componentes, por ejemplo, si un componente A depende de B1 y existe una alternativa B2, entonces en el grafo aparecerán conexiones desde A hasta B1 y desde A hasta B2.

Para predecir el futuro estado de cada uno de los componentes del sistema así como si tendrá la posibilidad de repararse encontrando alternativas, se ha diseñado un algoritmo de recorrido de grafos que al mismo tiempo utiliza técnicas de punto fijo.

El algoritmo recorre nodo a nodo y para determinar el estado operacional del mismo tiene en cuenta lo siguiente:

- El estado operacional de los nodos a los cuales se conecta.
- El tipo de relación que tiene con esos nodos (fuerte o débil)
- Si existen alternativas.
- Si el nodo con un estado operacional distinto de “Available” está tratando de repararse (porque esta en el estado “Stopped”).

Hay casos donde tales criterios no son evaluables debido a que no se puede obtener la información necesaria, por ejemplo, un componente que no tiene información adicional aparte del estado operacional como pudiera ser bases de datos, u otros Web Services ya que no tienen interfaz de gestión. En tales casos se toman valores por defecto.

Este recorrido se realiza una y otra vez hasta que en el grafo no hayan cambios.

5. Experimentación

Para la validación de la solución propuesta, se ha desarrollado por parte de los miembros del proyecto un prototipo de aplicación distribuida cuyo objetivo es dar soporte una red social de artistas gráficos (Comunidad de Artistas). A continuación se describe dicho prototipo y las medidas que se han obtenido en cuanto a esfuerzo de implantación de la auto-gestión con la infraestructura propuesta en este trabajo, además de otras pruebas de eficiencia en la ejecución.

5.1. Descripción del Prototipo

5.1.1. Funcionalidad

La aplicación proporciona soporte y otros servicios a una red social de artistas gráficos. La funcionalidad principal de esta aplicación es la siguiente:

- **Publicación de Trabajos:** Los artistas usuarios de la aplicación pueden publicar sus trabajos en la Web para que puedan ser accesibles a toda la comunidad de artistas.
- **Valoración de Trabajos:** Los artistas de la comunidad pueden a través de la Web hacer valoraciones a los trabajos publicados. Pueden comentarlos y darles una valoración por puntos.
- **Suscripciones:** Cada usuario de la Web, tiene la posibilidad de suscribirse a los trabajos de artistas concretos. El usuario recibirá notificaciones cuando los artistas publiquen trabajos nuevos.
- **Servicio de Resúmenes Semanales:** Los artistas usuarios de un servicio de envíos semanales de resúmenes del movimiento en la comunidad con respecto a las categorías a las que pertenezcan.

La interfaz gráfica es Web y el acceso a la misma es libre ya que tiene algunas funciones aunque restringidas para los visitantes, además, cualquiera pode darse de alta en ella y participar de los servicios que ofrece. En cuanto a las categorías del arte gráfico, se ha establecido las siguientes: Pintura, Dibujo y Fotografía. Pudiendo un usuario pertenecer a cualquiera de estas y publicar trabajos en cualquiera.

5.1.2. Casos de Uso

Los casos de uso más significativos se muestran en los diagramas de las Ilustraciones 39, 38 y 40. En ellos se describen aspectos como la exploración de los trabajos de los demás usuarios así como la posibilidad de valorarlos con puntuaciones y comentarios, la publicación y gestión de trabajos propios, y finalmente los de alta e inicio de sesión de usuarios; entre muchos otros que se detallan más adelante en el Anexo A.

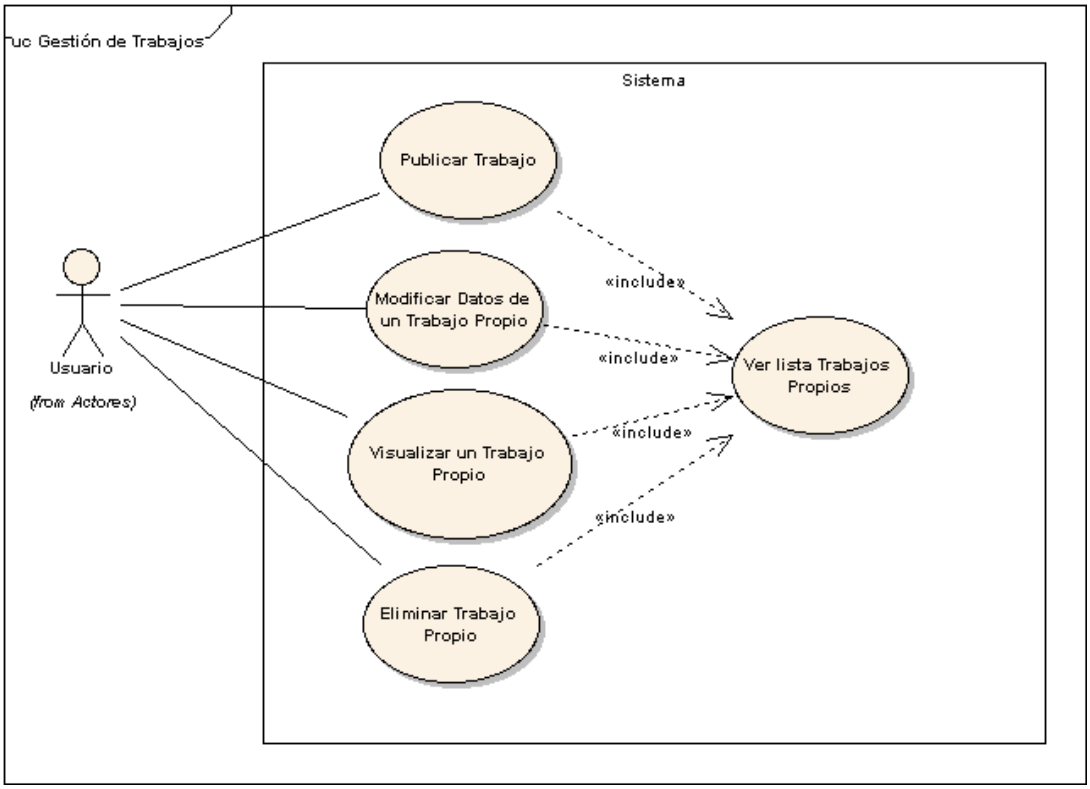

Ilustración 38: Casos de Usos de Gestión de Trabajos

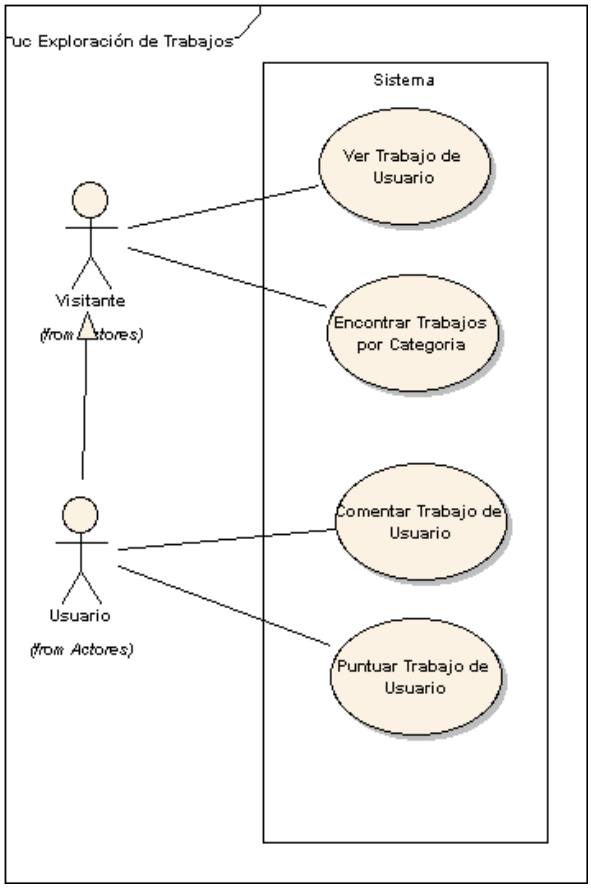

Ilustración 39: Casos de Usos de Exploración de Trabajos

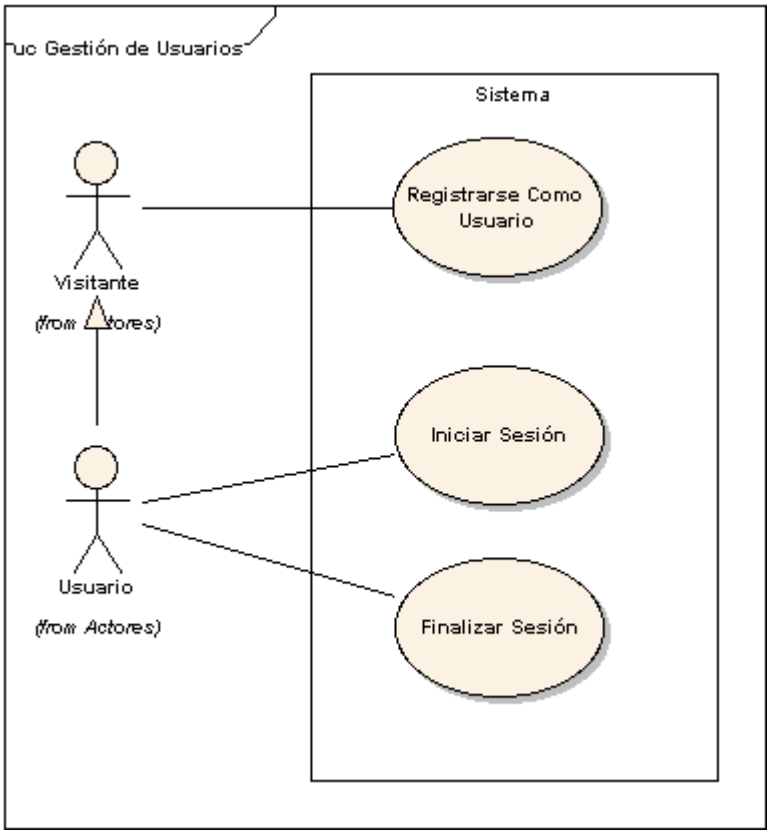

Ilustración 40: Casos de Usos de Gestión de Usuarios

La realización de los casos de usos en la fase de análisis ha servido para la identificación de los componentes que luego formaron parte de la arquitectura final de la aplicación, entre ellos están "ValoracionTrabajos", "GestionTrabajos", "ProcesamientoTrabajo", etc. Se han seleccionado tres ejemplos que se muestran en los diagramas "Comentar Trabajo de Usuario" en la Ilustración 41, "Publicar Trabajo" en la Ilustración 42, y "Registrarse como Usuario" en la Ilustración 43. Estos escenarios sirvieron para identificar los componentes principales de la arquitectura del prototipo.

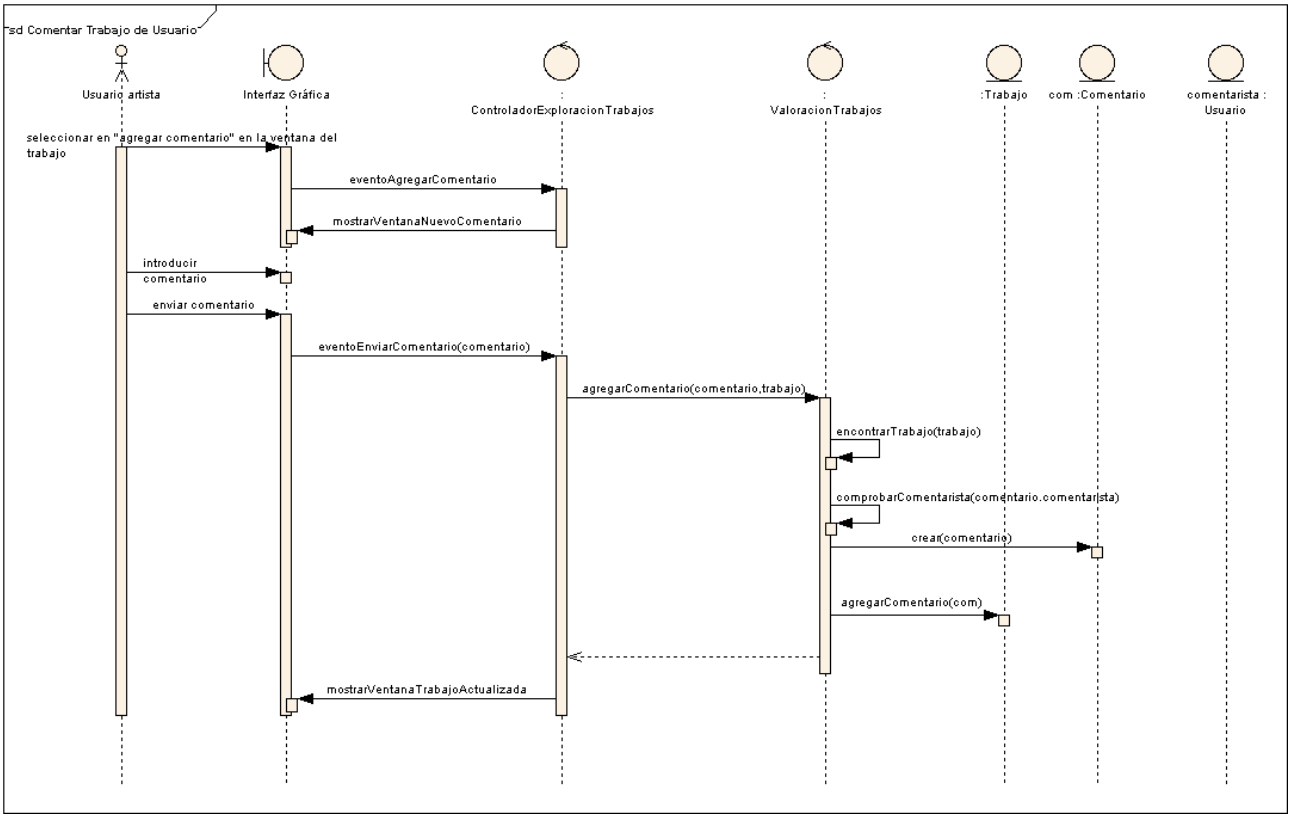

Ilustración 41: Realización del Caso de Uso "Comentar Trabajo de Usuario"

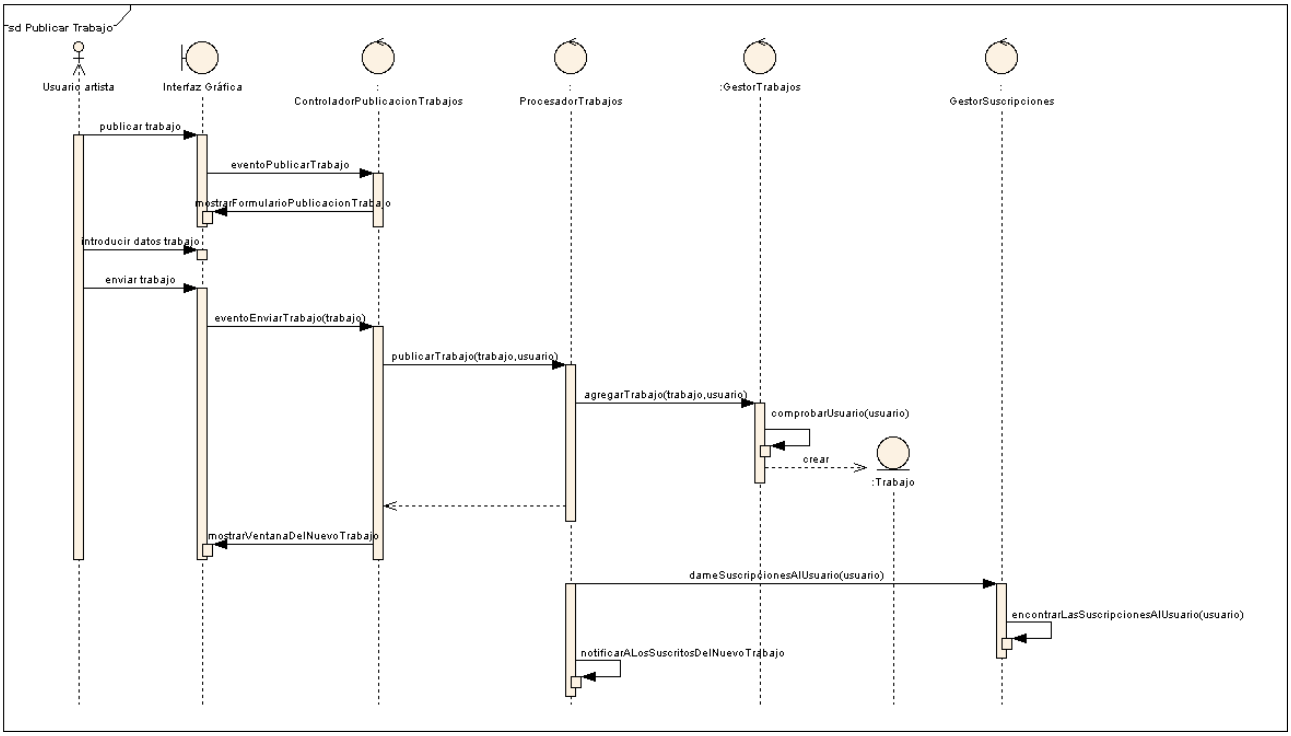

Ilustración 42: Realización del Caso de Uso "Publicar Trabajo"

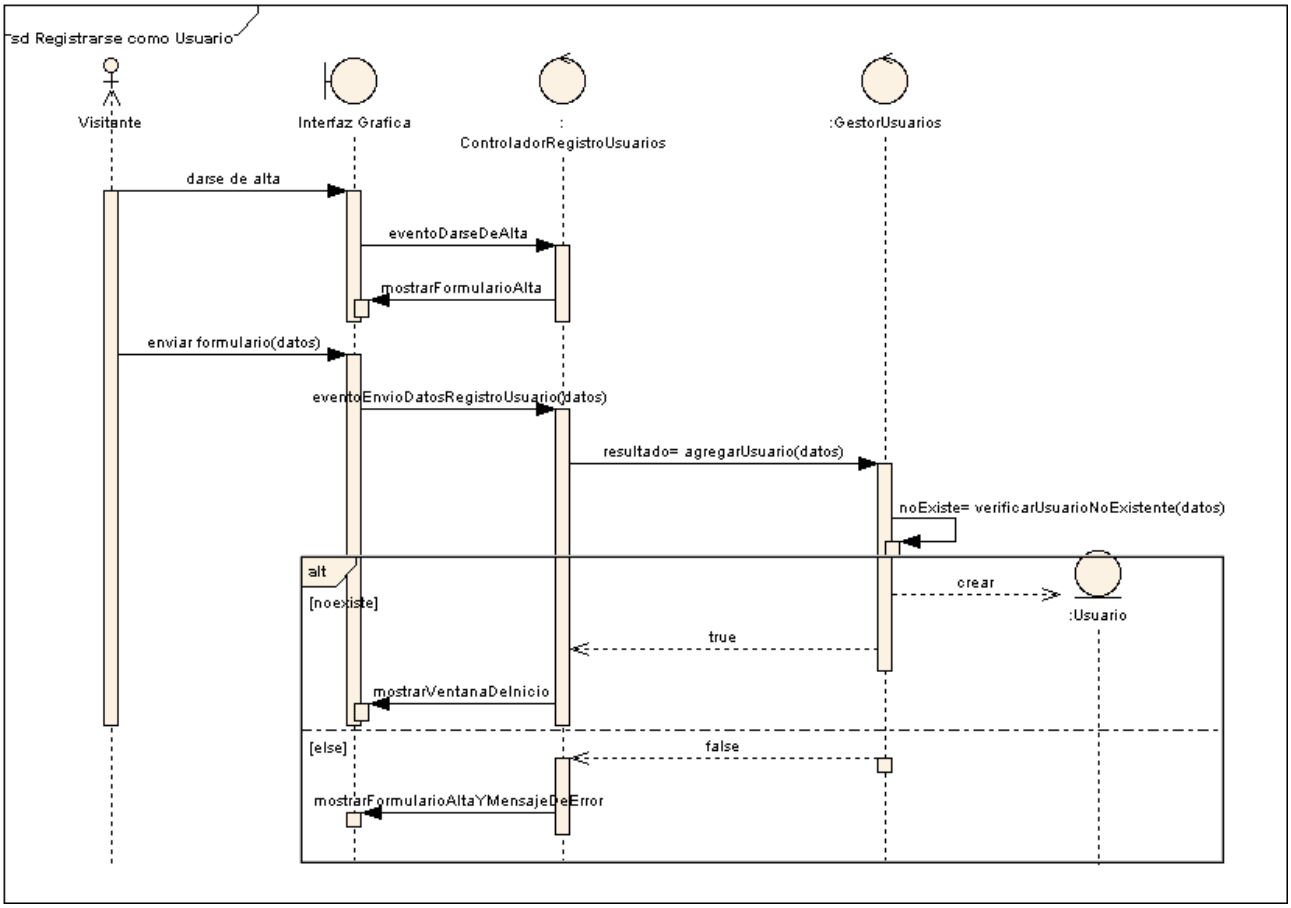

Ilustración 43: Realización del Caso de Uso "Registrarse como Usuario"

5.1.3. Arquitectura del Prototipo

La arquitectura del sistema de este prototipo esta formada por un conjunto de componentes distribuidos cuyas dependencias se describen en la Ilustración 44. La funcionalidad de cada componentes se describen en la Tabla 3.

Servicio	Interfaz	Descripción
Servicio de Gestión de Usuarios	IGestionUsuarios	Sirve como gestor de la entidad Usuario, es decir, brinda operaciones de dar de alta, dar de baja, consultas, modificaciones de usuarios.
Servicio de Gestión de Suscripciones	IGestionSuscripciones	Sirve como gestor de suscripciones, permite crear suscripciones, eliminarlas, etc...
Servicio de Gestión de Trabajos	IGestionTrabajos	Gestiona la entidad Trabajo en la base de datos, permite agregar un trabajo, eliminarlo, consultarlo y modificarlo.
Servicio de Resúmenes Semanales	IResumenesSemanales	Da el servicio de envío de resúmenes semanales a los usuarios. Permite activar y desactivar el servicio para un usuario concreto.
Servicio de Construcción de Resúmenes Semanales	IConstruccionResumenesSemanales	Construye los resúmenes que luego serán enviados a los usuarios.
Servicio de Procesamiento de Trabajos	IProcesamientoTrabajos	Procesas los trabajos para ser publicados, es decir, publica los trabajos y notifica a los usuarios suscritos que corresponda.
Servicio de Valoración de Trabajos	IValoracionTrabajos	Gestiona lo referente a la puntuación y comentarios de los trabajos.

Tabla 3: Servicios de la Aplicación

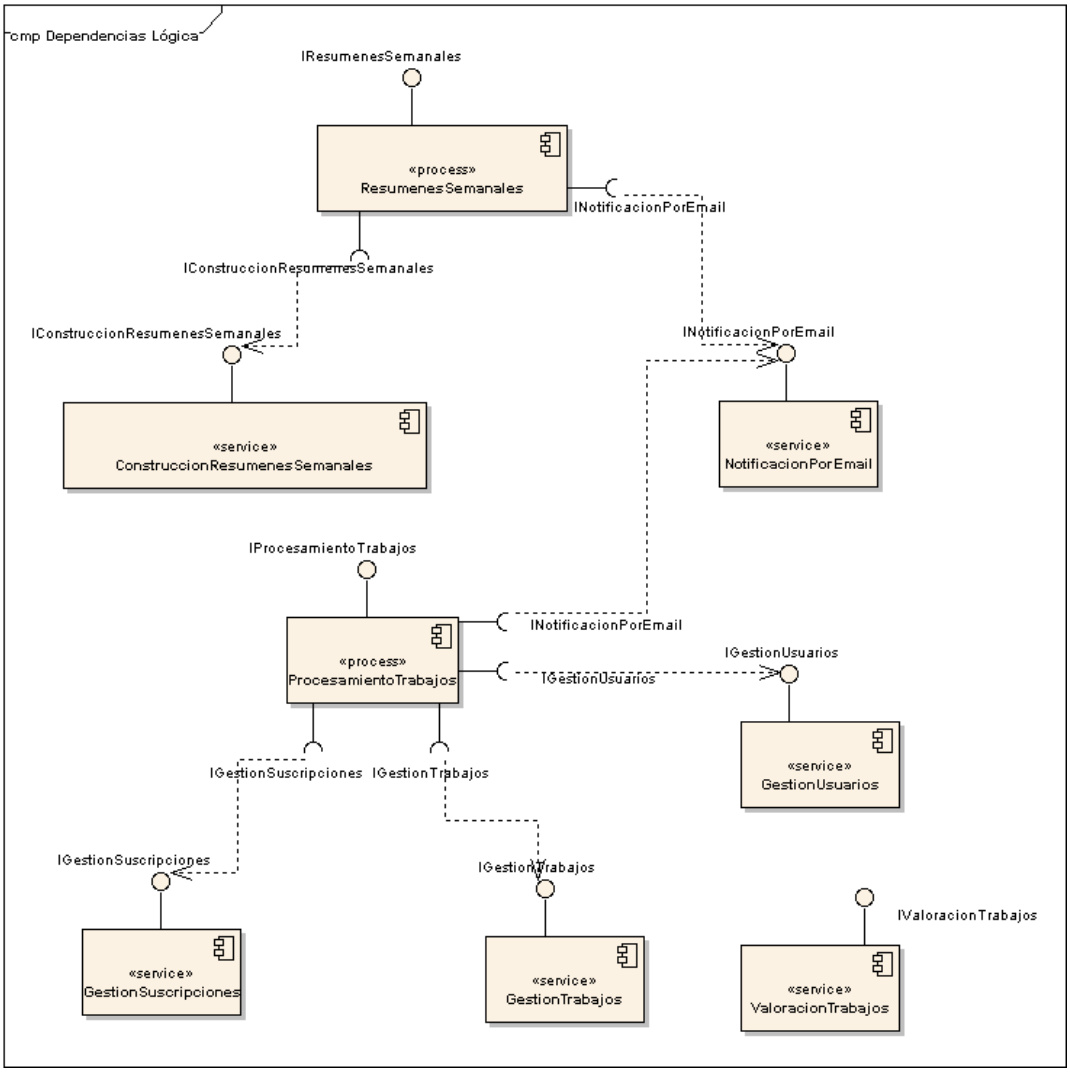

Ilustración 44: Módulos Principales del Prototipo

5.2. Utilización del Prototipo para la validación de la Infraestructura

5.2.1. Las pruebas

La validación de la Infraestructura se ha basado en la transformación de algunos componentes del prototipo para incluirles la capacidad de auto-gestionarse.

Se ha seleccionado dos componentes por sus características, estos son “GestionUsuarios” y “ProcesamientoTrabajo”. El primero se seleccionó por sus dependencias internas (ficheros de configuración de acceso a bases de datos), y el segundo por sus dependencias externas con otros componentes remotos, en concreto con el mismo “GestionUsuarios”.

Con estos componentes se validará desde el punto de vista técnico la monitorización y reparación en dependencias internas y externas, y también se validará la colaboración entres componentes auto-gestionados.

También se validará desde el punto de vista de ingeniería los costes en cuanto a esfuerzo y las ventajas en cuanto a la administración.

El proceso de transformación consistió en lo siguiente:

1. Crear los ficheros “iddf.xml” y “eddf.xml” e incluir la información de las dependencias.
2. Crear el fichero “mc-config.properties” e incluir la información de configuración de la infraestructura.
3. Crear el ProxyOfControl según recomendaciones.
4. Configurar los fichero “web.xml” para hacer accesible el punto de acceso de Gestión
5. Empaquetar el componente con los JAR de la Infraestructura.
6. Registrar en el Registro UDDI los puntos de acceso de Gestión de los servicios.

Téngase en cuenta que se parte de que en el Registro UDDI están publicados todos los servicios con sus correspondientes punto de acceso de uso.

Todo este proceso aplicado al componente “GestionUsuario” y “ProcesamientoTrabajos” tuvo un coste que se detalla en la Tabla 4.

Componente	Características	Esfuerzo
GestionUsuarios	Componente que tiene como única dependencia externa la base de datos, y como dependencias internas las librerías de “JPA” y el fichero “persistence.xml”.	2 Persona/Hora
ProcesamientoTrabajos	Componente que tiene Cuatro dependencias externas.	3 Personas/Hora

Tabla 4: Esfuerzos de Integración

A continuación se detalla el resultado de dos pruebas realizadas sobre dos escenarios concretos. Estos escenarios son:

1. **Vinculación dinámica:** Consiste en la vinculación del componente “ProcesamientoTrabajo” con un componente “GestionUsuarios” alternativo a causa de un fallo en el vinculado inicialmente.
2. **Reparación de Ficheros:** Consiste en la detección de permisos inapropiados (específicamente que no se puede leer) en el fichero interno “persistence.xml” del componente “GestionUsuario” , y en la reparación de los mismos (establecer los permisos de lectura).

El despliegue para ambas pruebas fue de la siguiente manera: se utilizaron dos PC's, en el primero (PC1) se desplegó el “ProcesamientoTrabajos” y en el segundo (PC2) se desplegó “GestionUsuarios” y “GestionUsuarios2” . La información técnica de ambos PC's se detalla en la Tabla 5

PC	Descripción Técnica Disponible
PC1	Procesador: 2.2 GHz Intel Core 2 Duo (dos núcleos) Memoria: 2GB 667MHz DDR2 SDRAM Caché de nivel 2: 4MB Velocidad del Bus: 800 MHz Tasa de transferencia del Disco Duro S-ATA: 1.5 Gb/s
PC2	Procesador: 1.73 GHz Intel Core Duo (dos núcleos) Memoria: 2GB 667MHz DDR2 SDRAM Caché de nivel 2: 2MB Tasa de transferencia del Disco Duro S-ATA: 1.2 Gb/s

Tabla 5: Descripción Técnica Disponibles de los PC utilizados para pruebas

En la Tabla 6 y la Tabla 7 se muestra el registro de los resultados de las pruebas.

Vinculación dinámica
I**nformación de acontecimientos:** *11:25:02* -> [ProcesamientoTrabajos] se detecta la caída mediante monitorización del componente GestionUsuarios . *11:25:07* -> [ProcesamientoTrabajos] se infiere y publica su nuevo estado operacional. *11:25:09* -> [ProcesamientoTrabajos] se comienza ejecución del plan de reparación que consiste en localizar una alternativa. *11:25:10* -> [ProcesamientoTrabajos] la vinculado al nuevo servicio GestionUsuarios2 ha culminado con éxito.

Tabla 6: Registro de los resultados de la prueba del escenario "Vinculación dinámica"

Reparación de Ficheros
Información de acontecimientos: *11:48:17* -> [GestionUsuarios] se detecta que el “persistence.xml” tiene permisos no validos *11:48:21.434* -> [GestionUsuarios] el plan de reparación está creado y está comenzada la ejecución *11:48:21.456*-> [GestionUsuarios] se ha detectado que el “persistence.xml” está con los permisos apropiados.

Tabla 7: Registro de los resultados de la prueba del escenario "Reparación de Ficheros"

Como se puede observar, en el primer escenario la recuperación se efectuó en 8 segundos desde que se detecto el fallo y en el segundo escenario se efectuó en 4,5 segundos aproximadamente. El momento en el que se detecta un fallo depende del valor que se le ha dado a los quantum's de monitorización en las configuraciones.

Estas pruebas se repitieron varias veces con el mismo entorno y los resultados son casi idénticos, por eso solo se ha mostrado el registro del sistema de solo un par de ellas.

5.2.2. Interpretación de las Pruebas

Un proceso de reparación de permisos de un fichero de configuración de un componente tiene aproximadamente un coste de 16 personas-horas (unos dos días para un técnico). Esto es debido a que cuando un componente desplegado tienen un fichero con permiso corruptos primeramente lo detecta el cliente del mismo por un error del servicio, a continuación se le comunica al responsable correspondiente y comienza la búsqueda de la raíz del error, y finalmente se detecta que son los permisos corruptos y se repara. Desplegar el componente con la infraestructura integrada hubiera sido un esfuerzo adicional de 3 personas-horas y la reparación hubiera sido automática en 4,5 segundos. Teniendo en cuenta que si se repite el error, el esfuerzo de integración no se suma, la ganancia en tiempo para el técnico sería del 100% pues el técnico no haría absolutamente nada, y la ganancia en tiempo de reparación sería del 99.9921%.

Otro caso sería el proceso de reparación del fallo de un componente distribuido del cual dependen otros componentes en un sistema. Este proceso puede tener un coste en esfuerzo de 24 personas-horas (unos tres días para un técnico). Partiendo de que existe alguna alternativa, el tiempo de reparación sería desde el momento en que se detecta por parte de algún cliente indirecto el fallo hasta que es cambiado manualmente el fichero de configuración de los componentes que usan al componente dañado para indicar la nueva alternativa a la cual vincularse. La reparación en este caso ha sido de 8 segundos en cada uno de los componentes afectados y el coste de implantación sería unos 2 o 3 personas horas por componente. La ganancia en tiempo para el técnico sería del 100% nuevamente, y la ganancia en tiempo de reparación sería del 99,9907%.

La transformación de un componente para que tenga la capacidad de auto-gestionarse tiene un esfuerzo de 2 o 3 personas-horas, y esto comparado con lo que cuesta desarrollar su funcionalidad por lo general (incluso de semanas) es despreciable. Un componente que es desarrollado en dos semanas por un equipo de tres ingenieros tendría un esfuerzo de 240 personas-horas. El incremento de esfuerzo que supone la transformación de dicho componente con la Infraestructura IADESS para hacerlo auto-gestionado sería del 1.25%.

El esfuerzo de reparar componentes que fallan una vez puesto en funcionamiento que puede ser varios días. El esfuerzo de desarrollar desde cero un sistema de detección de fallos y vinculado dinámico que puede ser de meses. Estos

esfuerzos son considerables en comparación con el hecho de transformar un sistema existente con la Infraestructura IADESS para dotarle de la capacidad de auto-gestionarse y también en comparación con el tiempo que demora la Infraestructura en reparar los fallos reparables por ella. Las ganancias en tiempo de reparación están en el orden del 99%, y el esfuerzo añadido que trae la transformación es del orden del 1.25% por cada componente. Esto es un dato a considerar por cualquier gestor de proyecto software.

Cada equipo de desarrollo debe hacer su análisis correspondiente para tomar un decisión así, ya que hay otros costes como el de aprendizaje de la Infraestructura y como el costes de eficiencia. Es evidentemente que introducir un sistema de gestión automático exige más memoria para la operatividad del componente.

6. Conclusiones

La complejidad de los nuevos sistemas software crece enormemente con el tiempo en número y tipos de componentes y relaciones. La inclusión de tecnologías distribuidas y de estándares relacionados hace posible la existencia de interoperabilidad a nivel global dejando libres a los desarrolladores de tratar con asuntos de compatibilidad de bajo nivel. Sin embargo, con el incremento de la complejidad de los sistemas aparecen nuevos tipos de problemas, y para enfrentar estos problemas de manera eficiente es necesario introducir automatización en la gestión de los mismos. Una posible solución podría consistir en habilitar a los sistemas para auto-gestionarse.

La principal aportación de este trabajo es la descripción completa de un modelo arquitectónico y de una infraestructura para soportarlo que consigue dotar de auto-reparación a un sistema distribuido basado en Web Services. El modelo, además, sigue abierto para la inclusión de mejoras, entre las que se encuentra la realización del resto de objetivos de auto-gestión, como se especifica en la sección 6.2 Adicionalmente, se ofrecen resultados de pruebas para la validación y otras aportaciones secundarias, que se describen en la sección 6.1.

También es interesante considerar las líneas alternativas que han ido surgiendo en el desarrollo del proyecto, las cuales ayudan a situar la opción escogida dentro del marco de todas las soluciones posibles que se pudieron obtener, dando así otra visión de los puntos fuertes y las deficiencias de la solución presentada. Estas alternativas se discuten con más detalle en la sección 6.3

Por último, como Proyecto de Sistemas Informáticos, de todo el desarrollo del trabajo se extraen una serie de conclusiones académicas que se describen en la sección 6.4.

6.1. Aportaciones

Las principales aportaciones del proyecto son las siguientes:

- Elaboración de un modelo arquitectónico para un sistema auto-gestionado.
- Desarrollo de una infraestructura para la aplicación de auto-gestión a un sistema existente.
- Desarrollo de una herramienta auxiliar de monitorización de un sistema auto-gestionado.
- Desarrollo de un prototipo de sistema basado en Web Services para aplicar la Infraestructura.
- Resultados de las pruebas de la Infraestructura con el prototipo.

El modelo arquitectónico desarrollado en este trabajo presenta varias ventajas respecto a las existentes en el dominio de los sistemas auto-gestionados: ausencia de puntos de fallo únicos, mayor flexibilidad y extensibilidad, así como mayor facilidad para dotar de capacidades de auto-gestión a aplicaciones existentes basadas en la tecnología de Web Services.

6.2. Trabajo Futuro

6.2.1. Auto-optimización, auto-configuración y auto-protección

Los siguientes pasos que deberían darse consisten en habilitar el sistema de gestión para la auto-optimización, auto-configuración y auto-protección. El último objetivo podría alcanzarse habilitando a cada componente para auto-protegerse individualmente -con la idea de que el sistema completo está protegido si cada parte está protegida, al igual que está reparado si cada parte está reparada. Sin embargo, los dos primeros objetivos podrían no ser alcanzables mediante el trabajo individual aislado de cada parte del sistema, ya que involucran un ajuste de parámetros a nivel de sistema que deben hacerse de manera consistente globalmente, de modo que cada componente debe “estar de acuerdo” sobre cada cambio que se lleve a cabo. Por estas razones se hace necesario la introducción de *coreografías* -tareas de grupo efectuadas por un conjunto de agentes que intentan alcanzar un objetivo común- y de un conjunto de protocolos que las soporten.

6.2.2. Generación automática de código

Otro paso interesante a dar es llevar a cabo la construcción de una herramienta que genere automáticamente las clases Proxy y los ficheros de configuración necesarios para la autogestión del sistema tan sólo observando algún tipo de anotaciones de los desarrolladores en el código del componente de negocio. Por ejemplo, la herramienta podría detectar el uso de Web Services externos u otros componentes ajenos si ese uso se declara previamente con la inclusión de anotaciones. Lo mismo puede hacerse para dependencias internas. La clase Proxy se haría de acuerdo a las interfaces de negocio implementadas por el componente. Este estilo de declaración de dependencias orientada a anotaciones es mucho más intuitiva y menos propensa a errores que codificar sin ayuda los ficheros de dependencia XML.

6.2.3. Autogestión de la autogestión

Finalmente, dado que el subsistema de autogestión es también un sistema, se le puede aplicar autogestión. Hacer esto puede ser útil para prevenir errores en las tareas de gestión y para asegurar que la maquinaria (ficheros de configuración y clases auxiliares) está lista. Además, esto es necesario para tener un sistema totalmente auto-gestionado.

6.3. Líneas Alternativas

Existen una serie de decisiones que se han ido tomando a lo largo del proyecto, que han marcado la ruta a seguir de la investigación. La mayoría de estas decisiones han sido obligadas por cuestiones de tiempo y recursos, y abren una serie de líneas alternativas que se podrían haber seguido y que aún podrían retomarse en estudios futuros.

Para empezar, se barajaron arquitecturas de tipo híbrido en las que se incluyesen ventajas de distintos tipos de arquitecturas, combinando sus diferentes estructuras de control. Sin embargo, se optó finalmente por uno de los extremos, el descentralizado total, con la intención de remarcar de manera más evidente las diferencias. Una vez observados los resultados de este trabajo, otra vía de investigación podría ir en busca de esta arquitectura híbrida.

Siguiendo con las decisiones, aunque se han seguido conceptos e ideas del estándar WSDM para especificar los interfaces Web Service de gestión de los componentes, éstos no siguen realmente el estándar de manera estricta, y por tanto no son compatibles desde un punto de vista de integración. La decisión de tomar esta vía ha estado marcada principalmente por las grandes dimensiones del estándar y la dificultad de aplicarlo estrictamente con las restricciones de tiempo. A pesar de todo, existen librerías y herramientas para aplicarlo al desarrollo de sistemas, pero aún están poco desarrolladas y no se adaptaban bien a las necesidades existentes. Una línea alternativa sería rehacer las interfaces de gestión siguiendo el estándar de manera formal.

Continuando, Java Management eXtensions (JMX) [11] es una tecnología que provee herramientas para construir soluciones para la gestión y monitorización de dispositivos y aplicaciones en un entorno web distribuido. Aún habiéndose considerado su existencia, no se ha investigado lo suficiente en este área, de la cual se podrían extraer quizá más medios para aplicar la infraestructura.

Para finalizar, existe también una tecnología, llamada Service Data Objects (SDO) [22] que permite el acceso a fuentes de datos vía Web Services de una manera uniforme, independiente de la tecnología subyacente. También sería un campo interesante de estudio, pero esta tecnología se desechó por cuestiones de ineficiencia.

6.4. Conclusiones Académicas

6.4.1. Conocimientos Practicados y Adquiridos

Los conocimientos que fueron aprendidos durante la carrera y practicamos fueron los siguientes:

- Ingeniería del Software (Gestión, Análisis y Diseño Orientado a Objetos).
- Inteligencia Artificial (Programación con Reglas, Agentes, Modelado de Comportamientos con Máquinas de Estados).
- Estructuras de Datos y de la Información (Lista, Colas, etc.., Aspectos de Eficiencia).

- Metodología y Tecnología de la Programación (Recorrido de Grafos, Aspectos de Eficiencia).
- Redes (Protocolo HTTP a fondo)

Los conocimientos adquiridos durante el proyecto fueron:

- Sistemas Distribuidos.
- Arquitecturas Orientadas a Servicios.
- Sistemas Multi-agentes.
- Programación con Reglas Orientadas a Objetos.
- La Auto-gestión de Sistemas Distribuidos.
- Presentación de Trabajos Científicos.

6.4.2. La Gestión del Proyecto

El proyecto se planteó con un proceso de desarrollo iterativo, abarcando todo el curso en tres grandes fases:

- **Preparación de los escenarios:** Desde el 15 de Octubre del 2007 hasta el 7 de Febrero del 2008 se desarrolló toda la aplicación Prototipo siguiendo un proceso de desarrollo basado en el Proceso Unificado, y simultáneamente se trabajo sobre el Estado del Arte.
- **Desarrollo de la Infraestructura:** A partir del 8 de Febrero del 2008 hasta el 28 de Marzo se desarrollo la infraestructura a partir de escenarios concretos de fallos creado en el mismo Prototipo. En esta fase se utilizó un Proceso Evolutivo, es decir, se fue aumentando la complejidad de la infraestructura poco a poco.
- **Documentación y Mantenimiento:** A partir del 29 de Marzo del 2008 hasta finalizar el Curso se trabajo sobre la Memoria del Proyecto y sobre el Artículo publicado en DCAI'08 que se muestra en el Anexo C, y también se corrigieron “bugs” de la infraestructura.

Las herramientas que dieron soporte al Proceso de Desarrollo fueron:

- Enterprise Architecture de Sparx Systems (http://www.sparxsystems.com) para el Análisis y Diseño en UML.
- NetBeans (http://www.netbeans.org) para la Implementación en Java y para las pruebas.

Y las herramientas que se usaron para dar soporte al Proceso de Gestión de las Configuraciones, específicamente para el Control de Versiones fueron:

- Subversion Server desplegado en Servidores de terceros como Assembla.com (http://www.assembla.com) y Google Code (http://code.google.com).
- Clientes de Subversion como TortoiseSVN (http://tortoisesvn.tigris.org), Svn Client (svn.tigris.org), La herramienta de SVN integrada en NetBeans y SCPlugin (http://scplugin.tigris.org).

Referencias

1. Apache Ant. URL: http://ant.apache.org.
2. Ana Mas. Agentes Software y Sistemas Multi-Agentes: Conceptos, Arquitecturas y Aplicaciones. apartado 2.3. Pearson Prentice Hall. 2005.
3. Ana Mas. Agentes Software y Sistemas Multi-Agentes: Conceptos, Arquitecturas y Aplicaciones. apartado 2.2.2. Pearson Prentice Hall. 2005.
4. Apache Struts 2.0. URL: .http://struts.apache.org/2.0.11.1/index.html.
5. Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence 19. pp 17-37. 1982.
6. Desmond F. D'Souza, Alan C. Wills. Objects, Components and Frameworks With UML. Addison Wesley, 1999.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software.
8. Garijo, F J., Bravo S., Gonzalez, J. Bobadilla E. BOGAR_LN: An Agent Based Component Framework for Developing Multi-modal Services using Natural Language. L.N.A. I, Vol 3040. Pp 207-. Springer-Verlag.2004.
9. IBM, An architectural blueprint for autonomic computing. URL: www-306.ibm.com/autonomic/pdfs/ACwpFinal.pdf.
10. IEEE Standard 1471.
11. Java Management eXtensions (JMX), URL: http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/.
12. Java Persistence API. URL: http://java.sun.com/javaee/technologies/persistence.jsp.
13. JBoss. Drools. URL: http://www.jboss.org/drools/
14. Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. Computer Magazine on January, pp. 41-50. 2003
15. Jonghyun Lee, Karpjoo Jeong, Hanku Lee, Inho Lee, Sangmoon Lee, Dosik Park, Changsung Lee, Woojin Yang. RISE: A Grid-Based Self-Configuring and Self-Healing Remote System Image Management Environment, Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06).
16. Kevin Lee, Rizos Sakellariou, Norman W. Paton and Alvaro A. A. Fernandes. Workflow Adaptation as an Autonomic Computing Problem. WORKS'07.
17. OASIS. UDDI v2. URL: http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2.
18. OASIS. WSDM Standard. An Introduction to WSDM. URL: http://docs.oasis-open.org/wsdm/wsdm-1.0-intro-primer-cd-01.pdf.
19. OASIS. WSDM Standard. URL: http://www.oasis-open.org/committees/wsdm/.
20. OASIS. Web Services Resource Framework (WSRF) TC. URL: http://www.oasis-open.org/committees/wsrf/.
21. P. Martin, W. Powley, K. Wilson2, W. Tian, T. Xu1 and J. Zebedee. The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services. International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07).
22. Service Data Object, URL: http://en.wikipedia.org/wiki/Service_Data_Objects.
23. Shameem Ahmed, Sheikh I. Ahamed, Moushumi Sharmin, and Munirul M. Haque. Self-healing for Autonomic Pervasive Computing. SAC'07.

24. The ICARO-T Framework. Internal report, Telefónica I+D May 2008.
25. W3C. WS-Addressing: Web Services Addressing. URL: http://www.w3.org/Submission/ws-addressing/.
26. W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction. URL: http://www.w3.org/TR/scxml/.
27. W3C. Web Service Management: Service Life Cycle. URL: http://www.w3.org/TR/wslc/.

A. Anexo: Descripción del prototipo de la Comunidad de Artistas

Especificación de los Casos de Usos

Se han especificado una serie de casos de usos para describir los requisitos de la aplicación.

Exploración de Perfiles.

Caso de Uso: Visualizar Perfil de un Usuario
ID:1
Breve Descripción: Visualización del Perfil de un Usuario Concreto.
Actores Principales: Visitante
Actores Secundarios:
Precondiciones:
Flujo Principal: 1. El Visitante selecciona en “Perfiles”. 2. El Sistema muestra una lista de usuarios registrados. 3. El Visitante selecciona a un usuario concreto. 4. El Sistema muestra el perfil del usuario seleccionado con la posibilidad de visualizar todos sus trabajos y de suscripción al mismo en caso de que sea un usuario el visitante.
Poscondiciones: El Sistema ha mostrado el Perfil que el Visitante deseaba visualizar.
Flujos Alternativos:

Caso de Uso: Suscribirse a las notificaciones de publicación de Trabajos
ID: 2
Breve Descripción: Suscripción a las notificaciones de publicación de Trabajos por parte de un usuario concreto.
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario a suscribirse ha iniciado sesión.
Flujo Principal: 1. El Usuario selecciona en “suscribirse a este usuario”. 2. El Sistema registra la nueva suscripción. 3. El Sistema indica al usuario que esta suscrito
Poscondiciones: Se ha registrado la nueva suscripción
Flujos Alternativos:

Exploración de Trabajos.

Caso de Uso: Ver Trabajo de un Usuario
ID:3
Breve Descripción: Visualizar un Trabajo concreto de un Usuario
Actores Principales: Visitante
Actores Secundarios:
Precondiciones:

Flujo Principal: 1. El Visitante selecciona “Perfiles”. 2. El Sistema muestra una lista de usuarios. 3. El Visitante selecciona un usuario concreto. 4. El Sistema muestra el Perfil de usuario seleccionado. 5. El Visitante selecciona “Ver Trabajos”. 6. El Sistema muestra la lista de todos los Trabajos publicados de este usuario. 7. El Visitante selecciona un Trabajo concreto. 8. El Sistema muestra detalles del Trabajo seleccionado.
Poscondiciones: El sistema ha mostrado detalles del Trabajo que el visitante deseaba visualizar
Flujos Alternativos:

Caso de Uso: Encontrar trabajos por categorías
ID:4
Breve Descripción: Encontrar trabajos por categorías
Actores Principales: Visitante
Actores Secundarios:
Precondiciones:
Flujo Principal: 9. El Visitante selecciona “Trabajos”. 10. El Sistema solicita una categoría concreta. 11. El Visitante selecciona la categoría. 12. El Sistema muestra la lista de Trabajos publicados de la categoría seleccionada con la posibilidad de acceder a los detalles de cada trabajo. 13. El Visitante selecciona el trabajo que desea visualizar. 14. El Sistema muestra los detalles de dicho trabajo.
Poscondiciones: El sistema ha mostrado detalles del Trabajo que el visitante deseaba visualizar
Flujos Alternativos:

Caso de Uso: Comentar un Trabajo
ID: 5
Breve Descripción: Agregar un comentario a un Trabajo concreto
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Visitante es un Usuario, es decir, esta registrado y tiene iniciada la sesión. El sistema esta mostrando los detalles del trabajo al que se le agregará un comentario.
Flujo Principal: 1. El Usuario selecciona en “Agregar Comentario”. 2. El Sistema muestra un editor de texto para que el usuario introduzca el comentario. 3. El Usuario introduce el comentario y selecciona en “Enviar”. 4. El Sistema agrega el comentario al Trabajo y muestra los detalles del mismo con el nuevo comentario agregado.
Poscondiciones: El sistema ha agregado el comentario al Trabajo.
Flujos Alternativos:

Caso de Uso: Puntuar Trabajo.
ID: 6
Breve Descripción: Asignarle una puntuación a un Trabajo
Actores Principales: Usuario

Actores Secundarios:
Precondiciones: El Visitante es un Usuario, es decir, esta registrado y tiene iniciada la sesión. El sistema esta mostrando los detalles del trabajo que se puntuará. El Usuario no ha puntuado anteriormente el Trabajo. (En el caso de que si haya puntuado anteriormente al Trabajo, esta opción esta deshabilitada en la GUI)
Flujo Principal: 1. El Usuario introduce la puntuación al Trabajo. 2. El sistema registra la puntuación asignada, y muestra los detalles del Trabajo con la puntuación media actualizada.
Poscondiciones: El sistema ha registrado la nueva puntuación. El Usuario ya no tiene la posibilidad de volver a puntuar el Trabajo.
Flujos Alternativos:

Gestión de Perfiles.

Caso de Uso: Modificar Perfil
ID: 7
Breve Descripción: Modificación de los datos del Perfil del Usuario.
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y tiene iniciada la sesión.
Flujo Principal: 1. El Usuario selecciona en “Mi Cuenta”. 2. El Sistema muestra el Perfil del Usuario en modo de Edición. 3. El Usuario cambia los datos que desea cambiar. 4. El Usuario selecciona en “Guardar Cambios”. 5. El Sistema registra los cambios y continua mostrando el Perfil del Usuario en modo de Edición pero actualizado.
Poscondiciones: El Sistema ha registrado los cambios del Perfil.
Flujos Alternativos:

Gestión de Suscripciones.

Caso de Uso: Visualizar las suscripciones del Usuario
ID: 8
Breve Descripción: Visualización de las suscripciones que tiene el propio Usuario a trabajos de otros.
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y tiene iniciada la sesión
Flujo Principal: 1. El Usuario selecciona en “Mis Suscripciones”. 2. El Sistema muestra la Lista de suscripciones del Usuario.
Poscondiciones:
Flujos Alternativos:

Caso de Uso: Eliminar suscripción
ID: 9
Breve Descripción: A partir de la lista de suscripciones, el Usuario elimina las que desee
Actores Principales: Usuario
Actores Secundarios:

Precondiciones: El Usuario esta registrado y tiene iniciada la sesión, además, se encuentra visualizando la lista de suscripciones
Flujo Principal: 1. El Usuario selecciona en “Eliminar” de una suscripción concreta. 2. El Sistema elimina la suscripción y muestra la lista actualizada.
Poscondiciones: El sistema ha eliminado la suscripción seleccionada.
Flujos Alternativos:

Gestión de Trabajos.

Caso de Uso: Ver lista de Trabajos propios
ID: 10
Breve Descripción: Visualización de la lista de Trabajos propios
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y tiene iniciada la sesión.
Flujo Principal: 1. El Usuario selecciona en “Mis Trabajos”. 2. El Sistema muestra la lista de trabajos del Usuario.
Poscondiciones:
Flujos Alternativos:

Caso de Uso: Eliminar Trabajo Propio
ID: 11
Breve Descripción: El Usuario decide eliminar un trabajo propio y lo elimina
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y ha iniciado sesión. El trabajo a eliminar existe.
Flujo Principal: 1. Incluido Caso de Uso 10 (Ver lista de Trabajos Propios.) 2. El Usuario selecciona en “Eliminar” de un Trabajo concreto. 3. El Sistema elimina el Trabajo seleccionado. 4. El Sistema muestra la lista de Trabajos propios actualizada.
Poscondiciones: El Trabajo esta eliminado.
Flujos Alternativos:

Caso de Uso: Modificar datos de un Trabajo Propio
ID: 12
Breve Descripción: Modificación de los datos de un Trabajo Propio, no se podrá eliminar el fichero, solo la información asociada (titulo y descripción)
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y tiene iniciada la sesión.

Flujo Principal: 1. Incluido Caso de Uso 10 (Ver lista de Trabajos Propios.) 2. El Usuario selecciona en “Editar” de un Trabajo concreto. 3. El Sistema muestra la Pantalla de Edición del Trabajo seleccionado. 4. El Usuario edita la información. 5. El Usuario selecciona en “Guardar Cambios”. 6. El Sistema registra los cambios. 7. El Sistema muestra la Pantalla de Detalles del Trabajo editado.
Poscondiciones: El Trabajo se ha modificado
Flujos Alternativos:

Caso de Uso: Visualizar un Trabajo Propio
ID: 13
Breve Descripción: Visualización de los detalles de un Trabajo concreto.
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y tiene iniciada la sesión.
Flujo Principal: 1. Incluido Caso de Uso 10 (Ver lista de Trabajos Propios.) 2. El Usuario selecciona en “Detalles” de un Trabajo concreto. 3. El Sistema muestra los detalles del Trabajo seleccionado.
Poscondiciones:
Flujos Alternativos:

Caso de Uso: Publicar un Trabajo.
ID: 14
Breve Descripción: Publicación de un Trabajo concreto.
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y tiene iniciada la sesión.
Flujo Principal: 1. Incluido Caso de Uso 10 (Ver lista de Trabajos Propios.) 2. El Usuario selecciona en “Cargar Trabajo Nuevo”. 3. El Sistema muestra el Formulario de Carga de Trabajo. 4. El Usuario introduce los datos del Trabajo y carga el fichero. 5. El Usuario selecciona en “Enviar Trabajo”. 6. El Sistema publica el Trabajo. 7. El Sistema muestra los detalles del nuevo trabajo publicado. El Sistema notifica a todos los usuarios suscrito a este Usuario del nuevo Trabajo publicado.
Poscondiciones: El nuevo Trabajo esta publicado, todos los usuarios suscritos están notificados del nuevo trabajo.
Flujos Alternativos:

Gestión de Usuarios.

Caso de Uso: Iniciar Sesión
ID: 15
Breve Descripción: Iniciar sesión en el sistema
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado

Flujo Principal: 1. El Usuario introduce el nombre de Usuario y la contraseña 2. El Sistema valida esta información. 3. El Sistema inicia la Sesión. 4. El Sistema muestra la Pantalla de inicio.
Poscondiciones: El Usuario ha iniciado la sesión.
Flujos Alternativos: 3ª. La información no es válida, El Sistema muestra una Pantalla de que los datos no son válidos.

Caso de Uso: Finalizar la sesión.
ID: 16
Breve Descripción: Finalizar la sesión del sistema
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y ha iniciado sesión.
Flujo Principal: 1. El Usuario selecciona en “Finalizar Sesión”. 2. El Sistema cierra la sesión del Usuario. 3. El Sistema muestra la Pantalla de inicio de Visitante.
Poscondiciones: El Usuario ha finalizado la sesión.
Flujos Alternativos:

Caso de Uso: Registrase como Usuario del Sistema
ID: 17
Breve Descripción: Registrarse como Usuario del Sistema.
Actores Principales: Visitante
Actores Secundarios:
Precondiciones:
Flujo Principal: 1. El Visitante selecciona en “Registrase”. 2. El Sistema muestra el formulario de Alta. 3. El Visitante introduce los datos solicitados y selecciona en “Enviar” 4. El Sistema verifica que no exista un usuario con el mismo nombre. 5. El Sistema registra el nuevo Usuario. 6. El Sistema muestra la Pantalla de Actualización del Perfil.
Poscondiciones: Usuario nuevo registrado
Flujos Alternativos: 5ª Existe un usuario con el mismo nombre, El sistema indica que ya existe un usuario con el mismo nombre y vuelve a mostrar el formulario de Alta (Paso 2).

Resúmenes Semanales.

Caso de Uso: Activación del Servicio de Resúmenes Semanales.
ID: 18
Breve Descripción: Activación del Servicio de Resúmenes Semanales que consiste en el envío semanal de un resumen con los 5 trabajos más puntuados dentro de cada categoría al correo electrónico del Usuario.
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y ha iniciado sesión.

Flujo Principal: 1. El Usuario selecciona en “Activar Servicio de Resúmenes Semanales”. 2. El Sistema activa el Servicio y muestra que esta activado al usuario.
Poscondiciones: El Servicio de Resúmenes Semanales esta activado y por lo tanto se envían resúmenes con los 5 trabajos más puntuados dentro de cada categoría al correo electrónico del Usuario con frecuencia semanal.
Flujos Alternativos:

Caso de Uso: Desactivación del Servicio de Resúmenes Semanales
ID: 19
Breve Descripción: Desactivación del Servicio de Resúmenes Semanales
Actores Principales: Usuario
Actores Secundarios:
Precondiciones: El Usuario esta registrado y ha iniciado sesión. El Servicio de Resúmenes Semanales esta activo.
Flujo Principal: 1. El Usuario selecciona en “Desactivar Servicio de Resúmenes Semanales”. 2. El Sistema desactiva el servicio y muestra al Usuario que esta desactivado.
Poscondiciones: El Servicio esta desactivado
Flujos Alternativos:

Glosario

Trabajo:	Se refiere a la obra artística que ha hecho un artista y que ha digitalizado para publicarlo en nuestro sistema.
Dibujo:	Es un arte visual en el que se utilizan varios medios para representar algo en un medio bidimensional con el uso de líneas. Los materiales mas comunes son los lápices de grafito, la pluma estilográfica, ceras (crayón), carbón, etc.
Fotografía:	La fotografía es la técnica de grabar imágenes fijas sobre una superficie de material sensible a la luz. En este caso, se trata la Fotografía como arte.
Pintura:	Es una de las Bellas Artes, por lo que se entiende como pintura artística al área en la clasificación clásica de las artes como el arte que trata la expresión empleando la teoría del color, por lo que se constituye como un arte primordialmente visual. Los instrumentos que se utilizan son: Pinceles, Brochas y Cuerdas.

Descripción de la Arquitectura

Para describir la arquitectura de la aplicación se ha procedido a seleccionar los puntos de vistas a enfocar [10].

Selección de los Puntos de Vista de la Arquitectura para su Descripción

Los puntos de vistas que se han seleccionado para describir la arquitectura son:

- **La Lógica:** Describe las entidades y los servicios de la aplicación.
- **La Estructura:** Describe la organización por paquetes del código
- **La Dinámica:** Describe la interacción entre las entidades y los servicios con el propósito de implementar los requisitos propuestos.
- **Los Componentes:** Describe los componentes del sistema, es decir, las interfaces que proveen y que requieren. También las dependencias que hay entre ellos
- **La Implementación:** Describe dónde están implementados los componentes.
- **El Despliegue:** Describe el modelo de despliegue de la aplicación.

La estructura de los paquetes en el prototipo que se ha desarrollado obedece a una arquitectura por capas la cual se muestra en la Ilustración 45.

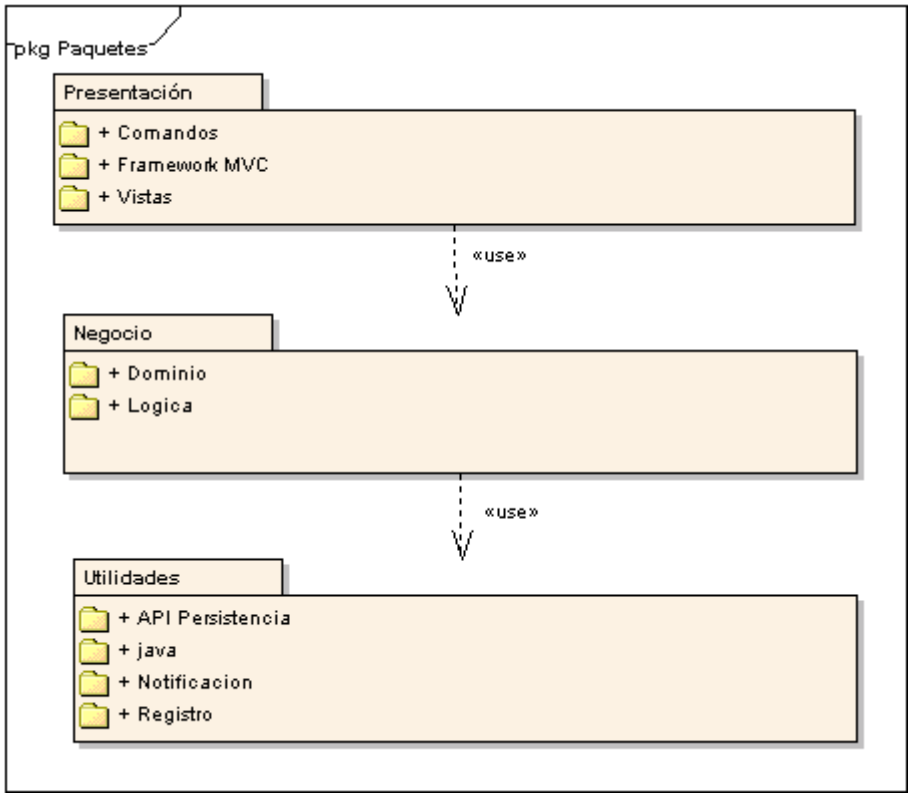

Ilustración 45: Paquetes en el Prototipo

Presentación: En este paquete se agrupan todas las clases relacionadas con la presentación de la información al usuario, a su vez, las clases se organizan en paquetes internos de acuerdo a su función, concretamente en Comandos (clases que son acciones) y Vistas (clases que representan ventanas).

Negocio: En este paquete se agrupan las clases que implementan la lógica del negocio en sí. Están las clases del Dominio y las clases de los servicios, estas últimas se encuentran dentro del paquete Lógica.

Utilidades: En este paquete se encuentra algunos subsistemas y APIs que la aplicación utiliza para sus funciones, entre ellas están frameworks de persistencia de objetos en base de datos SQL, servicios de Notificaciones por e-mail, servicios de Registros de interfaces, etc...

La Lógica

Modelo del Dominio:

En la Ilustración 46 se describe mediante un diagrama de clases las entidades del dominio, las cuales son persistentes. Estas se describen en la Tabla 8

Clase	Descripción
Usuario	Representa la cuenta del usuario de la aplicación
Perfil	Representa el perfil del usuario
Trabajo	Representa el trabajo del usuario
Categoría	Categoría de los trabajos y usuarios, estas pueden ser: Pintura, Dibujo y Fotografía
Suscripción	Representa una suscripción entre un suscrito y un publicador de trabajos
Comentario	Representa un comentario de algún usuario sobre algún trabajo
Puntuación	Representa la puntuación otorgada por algún usuario a un trabajo

Tabla 8: Descripción de las Entidades del Dominio

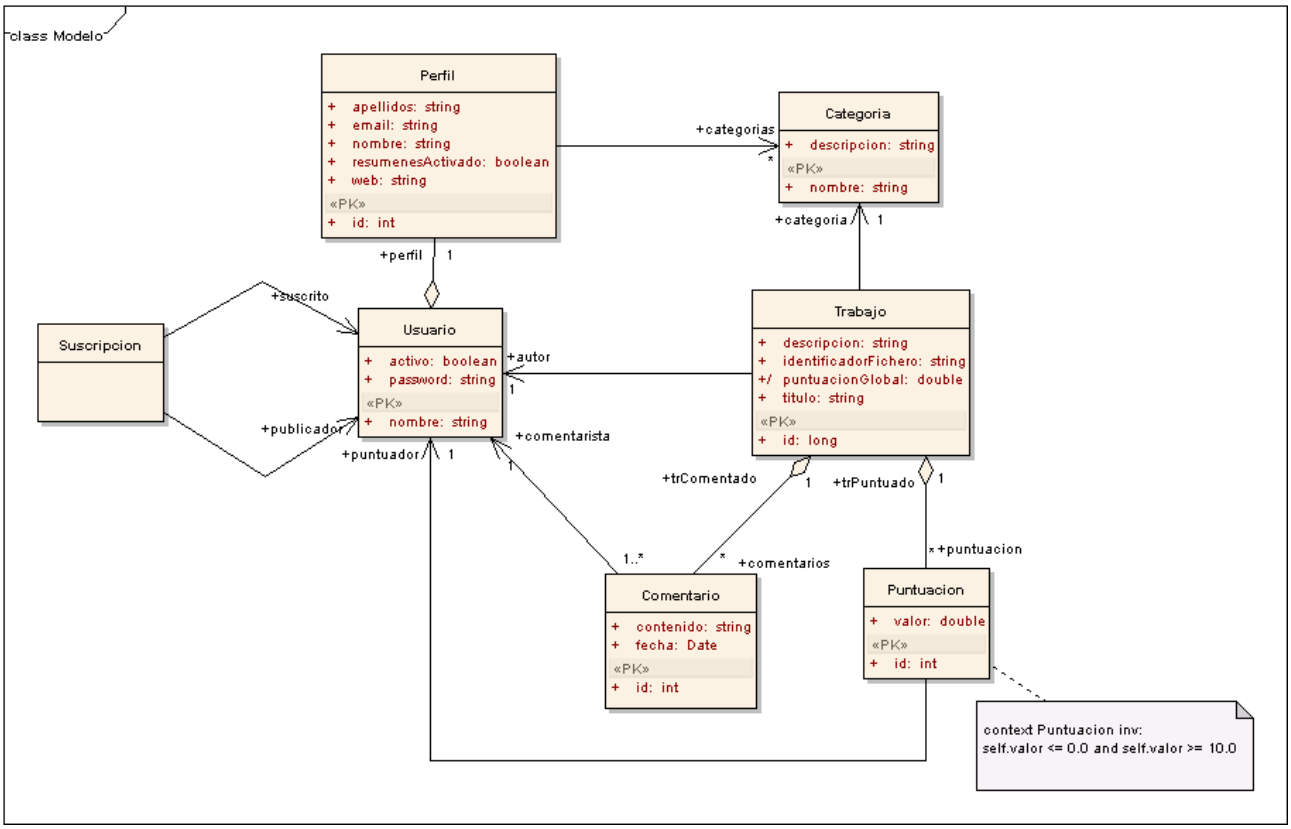

Ilustración 46: Modelo del Dominio

Servicios:

Los servicios de la aplicación se describen en la Tabla 9 y sus interfaces se detallen en el diagrama de la Ilustración 47.

Servicio	Interfaz	Descripción
Servicio de Gestión de Usuarios	IGestionUsuarios	Sirve como gestor de la entidad Usuario, es decir, brinda operaciones de dar de alta, dar de baja, consultas, modificaciones de usuarios.
Servicio de Gestión de Suscripciones	IGestionSuscripciones	Sirve como gestor de suscripciones, permite crear suscripciones, eliminarlas, etc...
Servicio de Gestión de Trabajos	IGestionTrabajos	Gestiona la entidad Trabajo en la base de datos, permite agregar un trabajo, eliminarlo, consultarlo y modificarlo.
Servicio de Resúmenes Semanales	IResumenesSemanales	Da el servicio de envío de resúmenes semanales a los usuarios. Permite activar y desactivar el servicio para un usuario concreto.
Servicio de Construcción de Resúmenes Semanales	IConstruccionResumenesSemanales	Construye los resúmenes que luego serán enviados a los usuarios.
Servicio de Procesamiento de Trabajos	IProcesamientoTrabajos	Procesas los trabajos para ser publicados, es decir, publica los trabajos y notifica a los usuarios suscritos que corresponda.
Servicio de Valoración de Trabajos	IValoracionTrabajos	Gestiona lo referente a la puntuación y comentarios de los trabajos.

Tabla 9: Servicios de la Aplicación

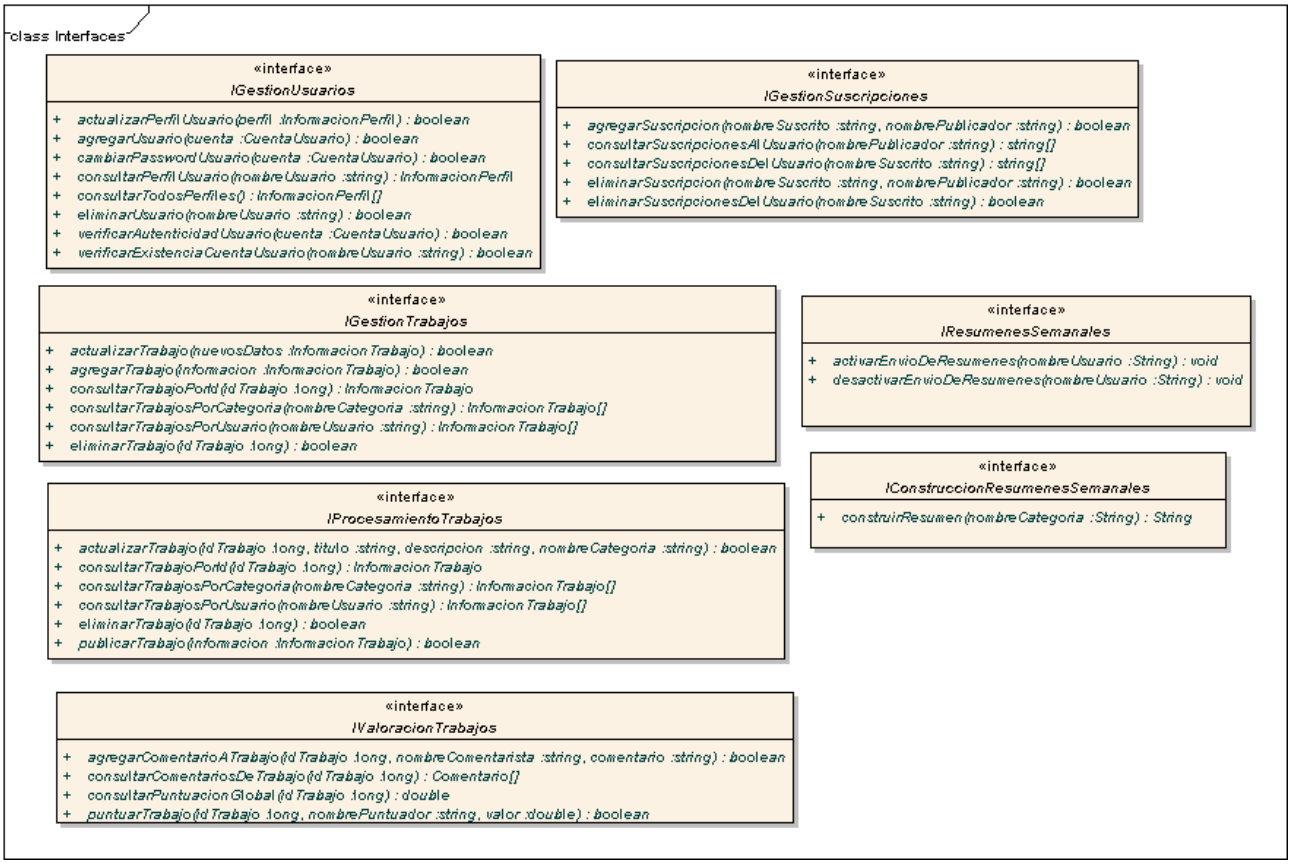

Ilustración 47: Interfaces

Las clases que encapsulan la información que requieren las operaciones de las interfaces de los servicios se describen en forma de diagrama de clases en la Ilustración 48

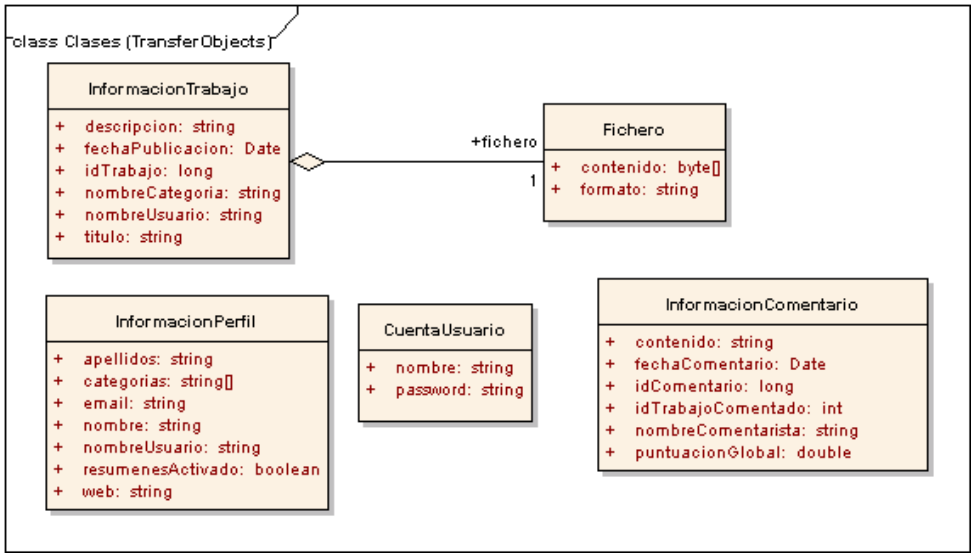

Ilustración 48: Objetos de Transferencia

Presentación:

La presentación de la aplicación se ha basado en un framework MVC que en este caso se ha decidido específicamente por Struts 2.0 [4].

En la Ilustración 49 se describe lo que se espera de dicho framework, en es la posibilidad de definir Ventanas y Acciones asociados a eventos.

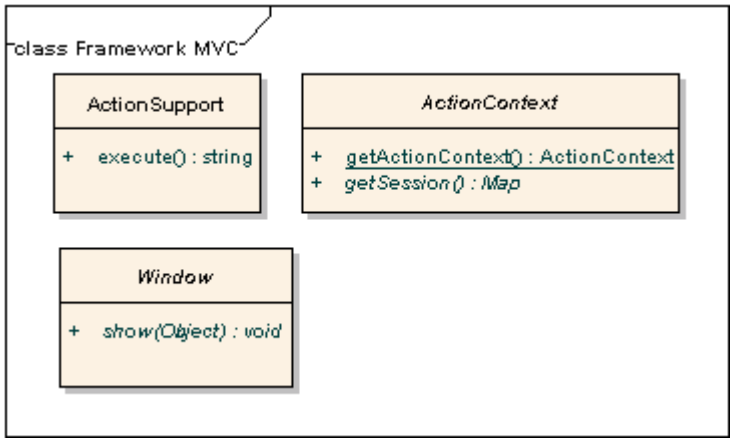

Ilustración 49: Framework MVC

Además de esto, también una especie de Contexto común de las acciones que tenga vida durante la sesión del usuario que funcione como Bus Común de Datos.

Utilidades:

En las utilidades necesarias se destacan dos en particular, la persistencia de objetos y la notificaciones por correo electrónico. Para la persistencia de objetos esperamos del framework lo descrito en la Ilustración 50.

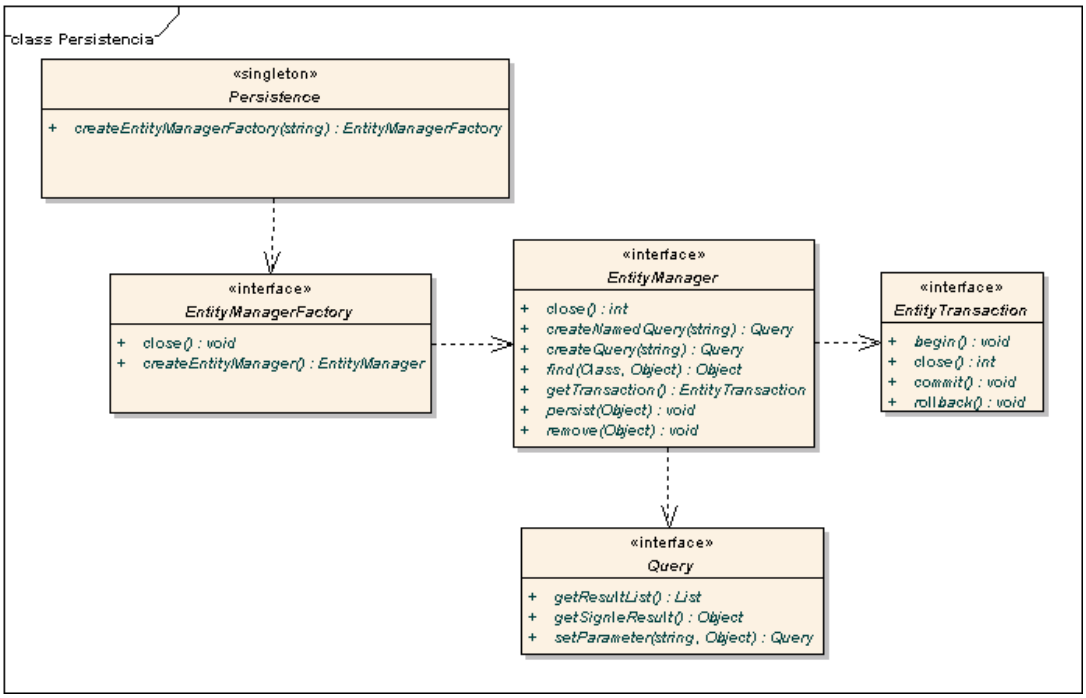

Ilustración 50: Framework de Persistencia

Es decir, se requiere de un Gestor de Entidades que tenga soporte de transacciones. Estas interfaces corresponden con la especificación JPA [12].

Para las notificaciones por correo electrónico se ha realizado un servicio concreto que se describe en la Ilustración 51.

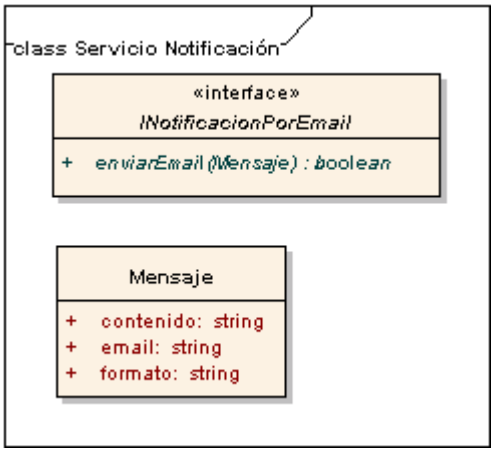

Ilustración 51: Servicio de Notificación por Correo Electrónico

La Dinámica

A continuación se describen las interacciones más significativas.

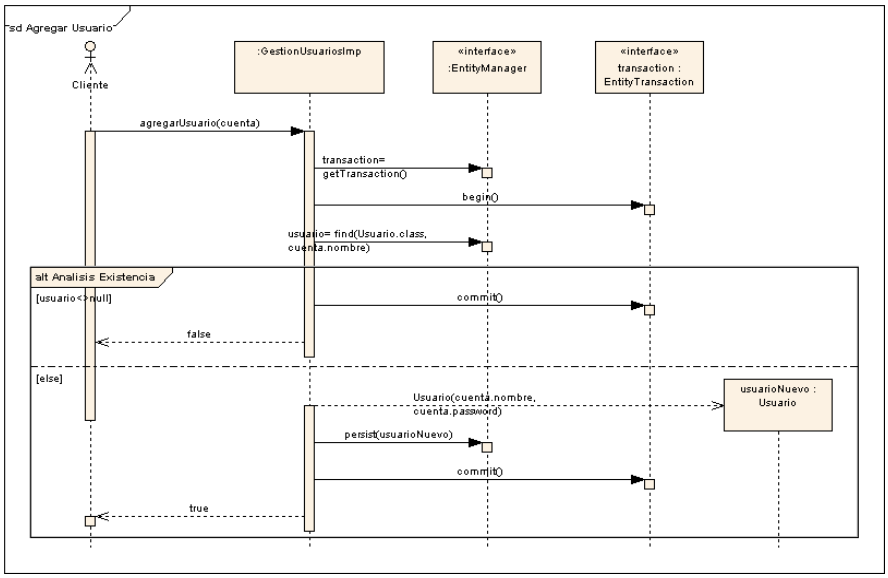

Ilustración 52: Diagrama Secuencia de Agregar un Usuario nuevo

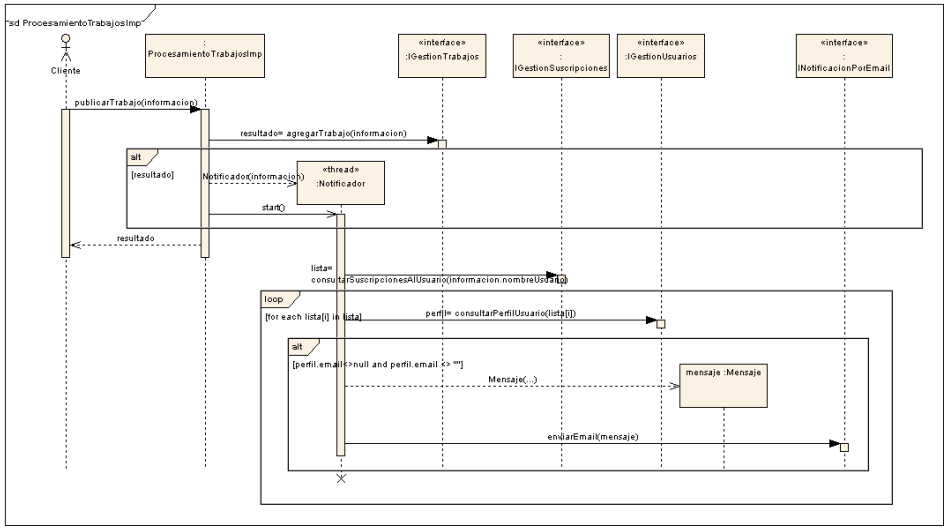

Ilustración 53: Diagrama de Secuencia de Publicar un Trabajo nuevo

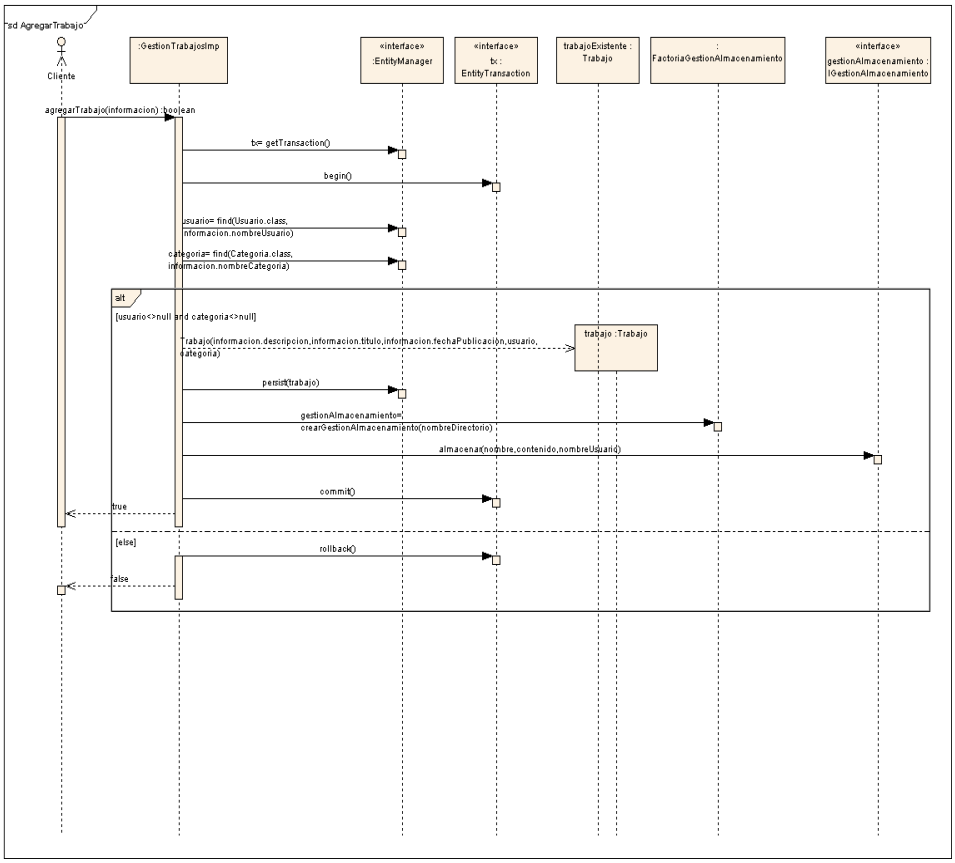

Ilustración 54: Diagrama de Secuencia de Agregar un Trabajo nuevo

Los Componentes

En el diagrama de componentes de la Ilustración 55 se muestra las dependencias entre los componentes lógicos implementados para esta aplicación. Estos componentes son los que ofrecen los servicios anteriormente descritos a través de sus correspondientes interfaces.

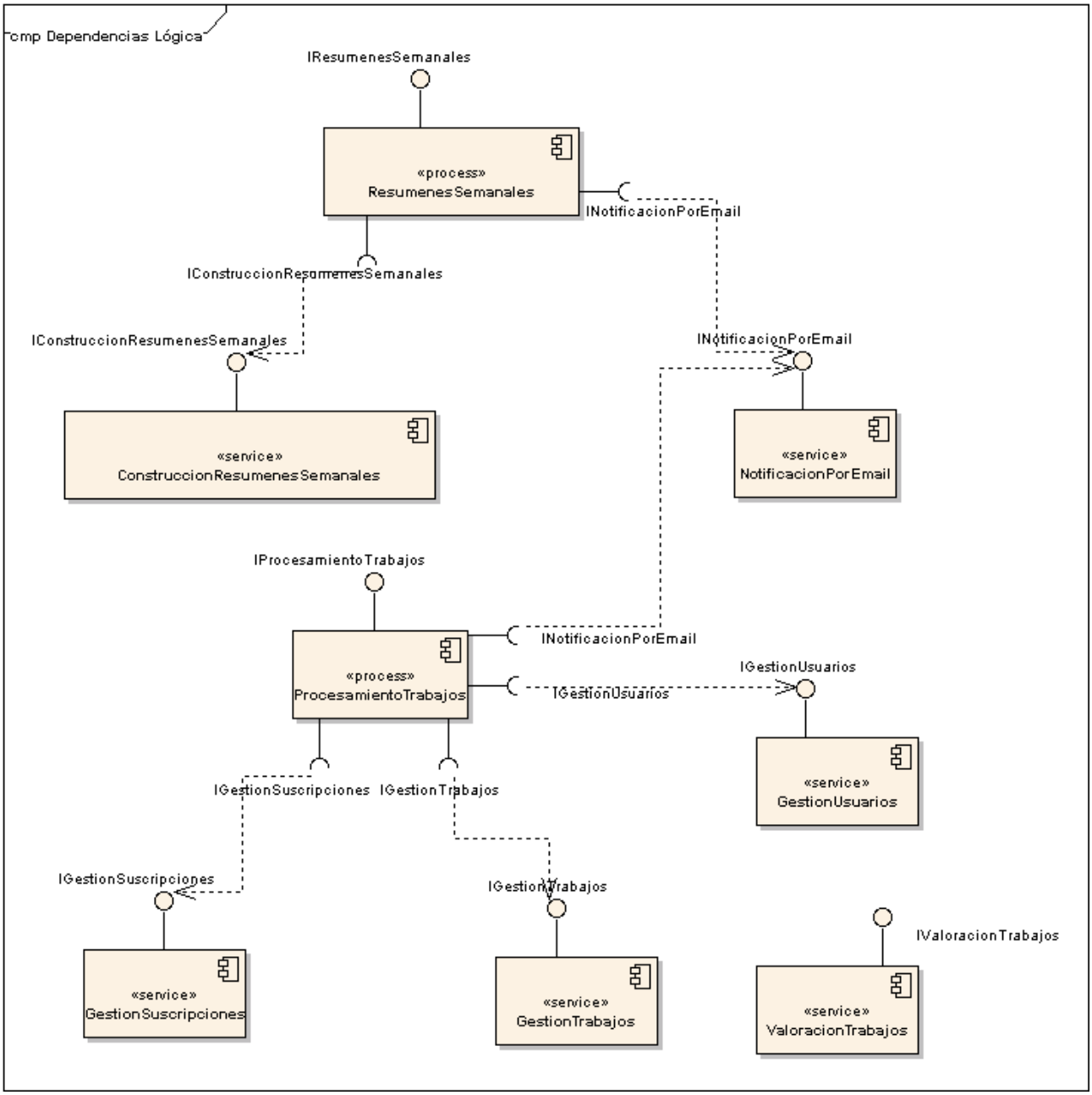

Ilustración 55: Dependencias Lógica entre Componentes

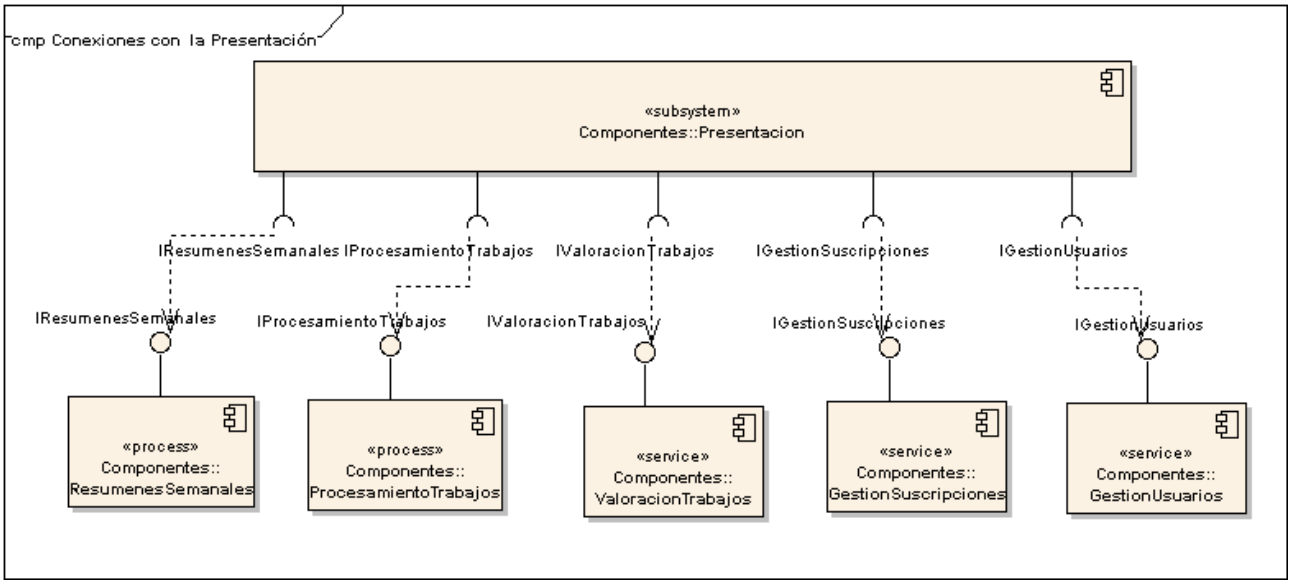

Ilustración 56: Conexiones entre la Presentación y el resto de Componentes

La Implementación

Los componentes se han empaquetados en artefactos WAR que serán desplegados en servidores de aplicaciones J2EE como Apache Tomcat. En la Ilustración 57 se describe que artefacto contiene que componente.

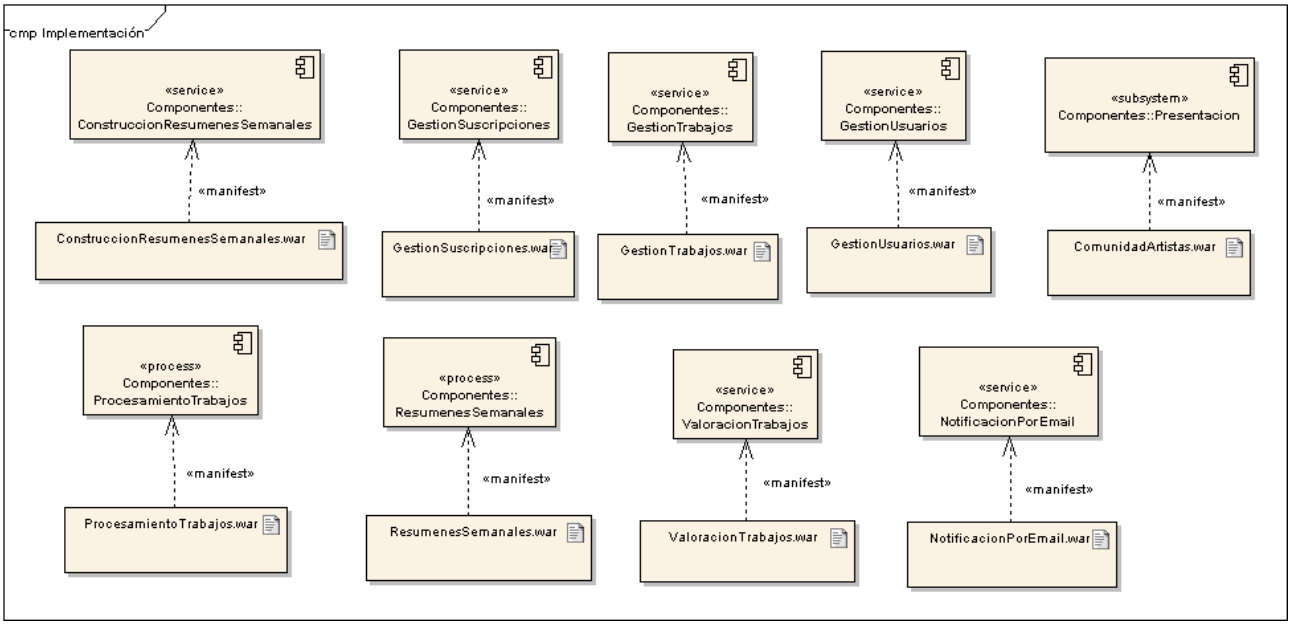

Ilustración 57: Implementación de los Componentes

El Despliegue

El modelo de Despliegue de la aplicación se muestra en la Ilustración 58. El componente “GestionUsuario” se encuentra replicado.

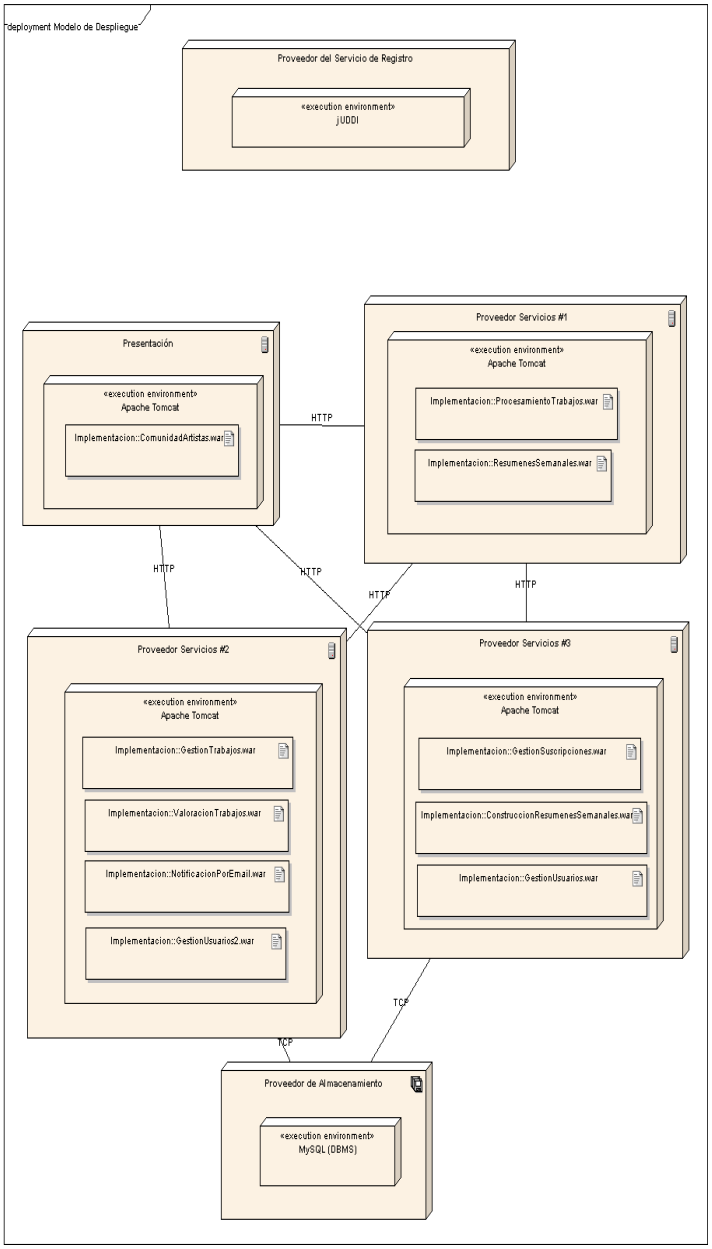

Ilustración 58: Modelo de Despliegue

Tecnologías

El prototipo hará uso de las tecnologías que se mencionan a continuación, junto con la justificación de tal decisión.

Ant: Apache Ant[1] ha sido seleccionada como herramienta para la gestión de las fuentes en nuestro proyecto.

Esta herramienta, hecha en Java, tiene la ventaja de no depender de las órdenes de shell de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas, siendo idónea como solución multi-plataforma.

Es una herramienta con muchísima madures, y muy probada, pues su uso es muy extendido.

Struts 2.0: Apache Struts [4] es una herramienta de soporte para el desarrollo Web guiado por el patrón MVC. Ha sido seleccionada debido a que ha sido utilizada por lo miembros del proyecto en ocasiones anteriores con éxito.

Web Service (SOAP, WSDL, UDDI): La tecnología Web Services como framework de comunicación entre componentes software es ampliamente conocida y su implantación en muchas soluciones ha propiciado la aparición de muchos framework de soporte y de mucha literatura al respecto.

También, esta tecnología en si misma, tiene características de interoperabilidad y de multiplataforma que añaden a las arquitecturas software que la usan.

Los miembros del proyecto ya han experimentado con la misma con resultados exitosos.

Además de estas cosas, es la base tecnológica de la propuesta descrita.

Todo esto nos llevo a decidir por esta.

JPA : Java Persistence API [12], más conocida por su sigla JPA, es la API de persistencia desarrollada para la plataforma Java EE e incluida en el estándar EJB3. Esta API busca unificar la manera en que funcionan las utilidades que proveen un mapeo objeto-relacional. El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos, y permitir usar objetos regulares (conocidos como POJOs).

Las implementaciones más populares son Hibernate, TopLink Essentials y OpenJPA.

Básicamente quita el esfuerzo del desarrollo de DAOs para el asunto de la persistencia de objetos. Además de esto, los miembros de proyecto tienen algo de experiencia en la utilización de la misma.

B. Anexo: ICARO-T 4.1

La Infraestructura Ligera de Componentes Software Java basada Agentes y Recursos y Organizaciones (ICARO-T) es un proyecto de Telefónica I+D, con el objetivo facilitar el desarrollo de aplicaciones distribuidas basadas en Agentes.

De esta Infraestructura se ha tomado para el desarrollo de los agentes el “patrón” de agente reactivo. Con este patrón se puede desarrollar un agente reactivo a partir de su comportamiento descrito como una máquina de estados y sub-estados.

Las interfaces de las que se disponen están descritas en la Ilustración 59, y el lenguaje mediante el cual se describen las máquinas de estados es el SCXML 1.0 [26].

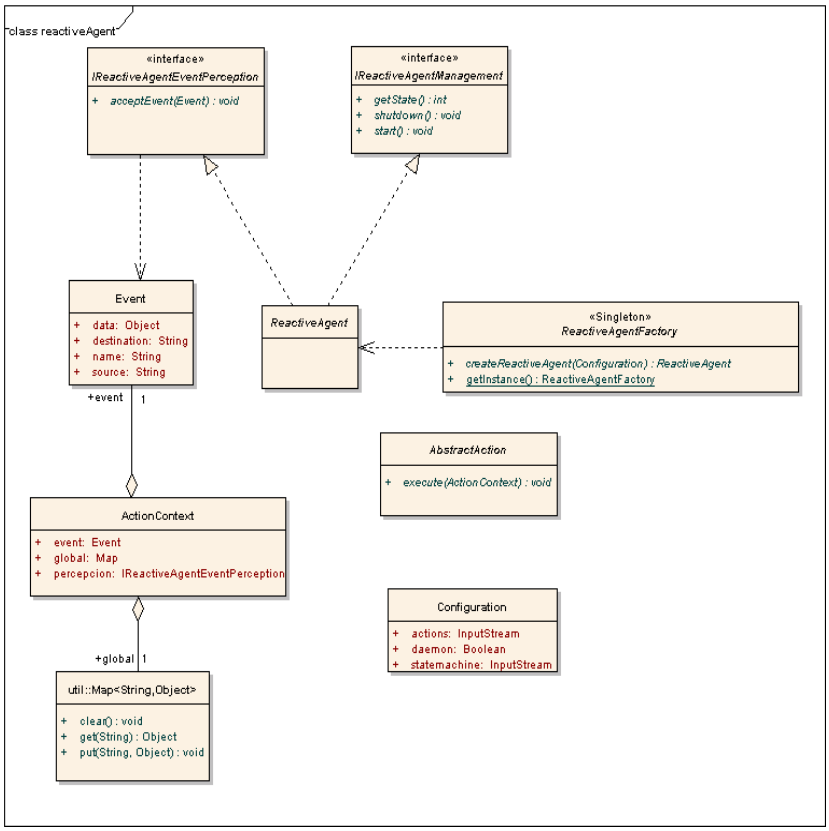

Ilustración 59: Patrón de Agente Reactivo

C. Anexo: Artículo aceptado para publicación en International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI'08)

Correo electrónico de aceptación del Artículo enviado:

Asunto: DCAI'08 Decision on your paper
Fecha: Tue, 17 Jun 2008 21:51:42 +0200
De: DCAI <dcai@usal.es>
Responder a: dcai@usal.es
Para: José María Fernández De Alba <jmfernandezdalba@gmail.com>

Dear José María Fernández De Alba,
You submitted a paper to be considered for publication in the InternationalSymposium on Distributed Computing and Artificial Intelligence 2008 (DCAI'08).

Details of the paper are as follows:
Title: A decentralized model for self-managed Web Services applications
Authors: José Mª Fernández de Alba, Carlos Rodríguez, Damiano Spina, Juan Pavón, Francisco J Garijo, Telefónica I+D
We are pleased to inform you that your paper has been accepted, subject to certain specified amendments being satisfactorily completed.

Enlace a página web del congreso donde se indica que los artículos aceptados serán publicados en la prestigiosa serie the prestigious *Advances in Soft Computing Series* de **Springer** Verlag:

http://dcai.usal.es/index.php?option=com_content&task=view&id=2&Itemid=3

A decentralized model for self-managed Web Services applications

José Mª Fernández de Alba, Carlos Rodríguez, Damiano Spina, Juan Pavón, Francisco J Garijo

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense Madrid
Facultad de Informática, Ciudad Universitaria s/n, 28040, Madrid, Spain
jpavon@fdi.ucm.es;fgarijo@tid.es

Abstract. Self-management in distributed systems is a way to cope with the growing complexity of these systems today, and its support in existing systems requires some transformation in their architectures. This paper presents a decentralized model for the implementation of self-management capabilities, which has also the advantage of avoiding the single point of failure (SPOF) issue, providing more robustness to the management system. The proposed architecture, which is described, in this paper, has been validated in a real distributed application.

Keywords: self-management, autonomic computing, multi-agent system, social network.

1 Introduction

Autonomic Computing is a concept initially developed by IBM [1], with the intention to cope with the increasing complexity of systems, as they grow in the number of elements and information. This solution intends to automate many of the functions associated with computing. In concrete, as [1] specifies, a computing system has the autonomic capability if it can manage itself only with the high-level objectives from administrators.

The goals of such *self-management* ability are to free system administrators from the details of system operation and maintenance while providing users with a high-performance service.

The four main aspects of self-management are: self-configuration, which implies automated configuration following high-level policies; self-optimization, which implies the system looking for the opportunity to tune performance; self-healing, which implies the detection and repairing of problems; and self-protection which implies the system protection against attacks and cascading errors.

IBM proposed a layered architecture [11] where the upper layers contained the autonomic managers, and the lowest layer is populated by the managed resources. The management interfaces of these resources are encapsulated as service endpoints, so that they can be accessed via an enterprise communication technology, like Web Services. The recommended management model is Web Service Distributed Management (WSDM) standard [3]. Autonomic Managers

(AM) in the control layer, are cooperating agents [5], which achieve their management goals following high-level policies. AMs share a knowledge base, which provide a common domain model and high-level policies.

The WSDM specification [3] enables management-related interoperability among components from different systems and facilitates integration of new components, improving scalability and flexibility. It also provides mechanisms for analyzing proactively different component properties such as quality of service, latency, availability, etc. An example of implementation of WSDM for Web Services is described in [14]. This model is based on the IBM approach using a centralized architecture with a common Knowledge Repository.

Other self-management architectures like RISE [12] are domain-specific. They focus on particular aspects such as: i) self-configuring and self-healing of remote system image, ii) workflow adaptation as an autonomic computing problem [13] and iii) self-healing problem for autonomic pervasive computing [15].

The work presented in this paper proposes a framework for incorporating self-management capabilities into Web Services applications using the WSDM model. This framework provides Web Services and Web Applications with autonomous features such as fault diagnosis, dynamic rebinding, file restoring, and resource substitution.

The approach consists on making each WS component of the system Self-Managed. Instead of having a common Knowledge Repository, which is often a Single Point Of Failure (SPOF), each self-managed Component has self-knowledge about its own dependences, and social knowledge about their dependent components. The aim of the paper is to describe the proposed approach, which is illustrated with a working example of self-healing. The validation framework is based on a website supporting a distributed social network for artists.

The plan of the paper is the following: the architectural approach is presented in section 2. A more detailed description of this architecture, including internal components and behaviour, is in section 3; and the planning model is in section 4. The case study and the validation of the proposed approach are in section 5, and finally a summary of the work done and future work are discussed in the conclusions.

2 Approach for transforming Systems based upon Web Services in order to enable self-management features

The proposed approach focus on distributed systems based upon Web Services technology. These systems could be transformed into self-management systems by applying a self-management framework to each component. The basic idea is to make each system component (WS), self-managed, by enhancing them with new management components implementing self-management capabilities. Then make the self-managed components cooperate in order to make the overall system self-managed. Figure 1, gives and example of transformation based on the studied case.

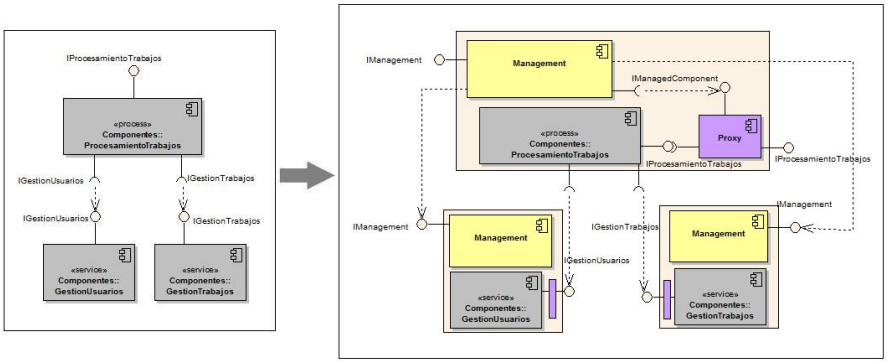

Fig. 1. Transforming a system into a self-managed system.

Each component has internal parts like files, libraries, etc., and possibly dependences with other components and servers. These components will be monitored, analyzed and controlled to provide self-management capabilities for each component and the whole system.

3 Self-managed Architecture

Figure 2 illustrates the internal structure of a service component. The "Component" is the original component that provides the logical operation of the service. The "Management" and "ProxyOfControl" components implement the management capabilities and are added to build the "NewComponent", which now has the original logical operation and self-management capability.

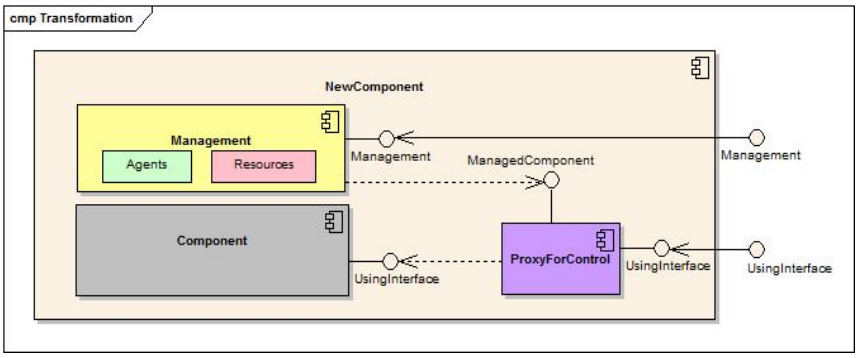

Fig. 2. Self-management component Architecture

The Management Component is made of packaged agents, resources and a model, which will be described later. The Management Interface offers operations to others self-managed components, which might use them to know its operational status.

The “ProxyOfControl” component controls the access to the managed component, avoiding possible misuses in inappropriate states, and providing information about the state of the managed component by catching Technical Exceptions. This component was designed using the State Design Pattern [10].

3.1 Modelling dependencies

Achieving self-management capabilities require a conceptual model of the domain involved representing explicitly the dependencies among components [2]. Figure 4 shows the model of dependencies shared by the management components.

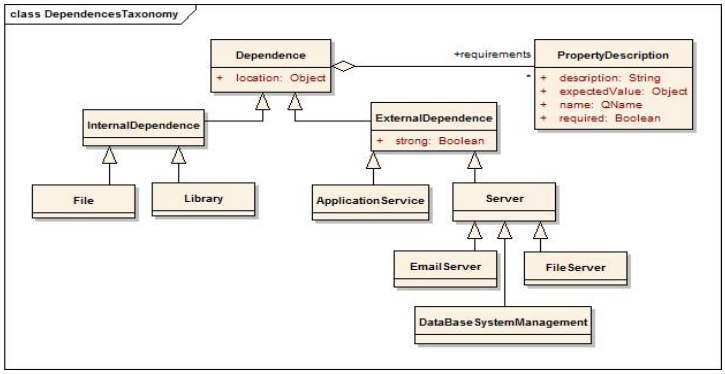

Fig. 4. Dependences

A managed component could have internal or external dependences: an internal dependence might be a dependence with a computing entity such as a file or a library, while an external dependence might be a dependence with a server, e.g. an email server, a database server, a file server or any application service. The application service external dependence refers to an abstract service, it means, a required interface, which is resolved at runtime.

All dependences have a location. The location is an attribute indicating how to access the component that supplies the dependence, for usage or monitoring purpose. For application services, the location refers to the required interface, and the registry service location to find out a particular service to supply the dependence.

The dependence also has a set of requirements that define the properties to be satisfied by the related component.

A property description has the following attributes:

- Name: a full-qualified name.
- Description: a human-readable description.
- Required: if it is required or optional.
- Expected value: the expected value of the property.

Examples of properties are: can read, can write, XML well-formed, syntax of content validated, file names patterns, availability, time of response, etc.

3.2 Self-management Agents

The logical control was designed using the Multi-Agents paradigm [5], and it is implemented using component patterns based on the ICARO-T framework [7] [8]. There are four types of agents:

- **The Manager:** It is responsible for creating, terminating and management the rest of agents and resources in the Management Component.
- **The Installer:** It is responsible for verifying the internal dependences and to fix possible errors that may happen during the first execution of the managed component.
- **The Runtime agent:** It is responsible for verifying the right functioning of external dependences at runtime, passing control to the Repair agent when errors occur.
- **The Repair agent:** It has the responsibility of fixing errors send by the Runtime agent.

3.3 Resources

Resources perform different task required by the agents. Some of them are responsible for monitoring the properties of the managed component's dependences.

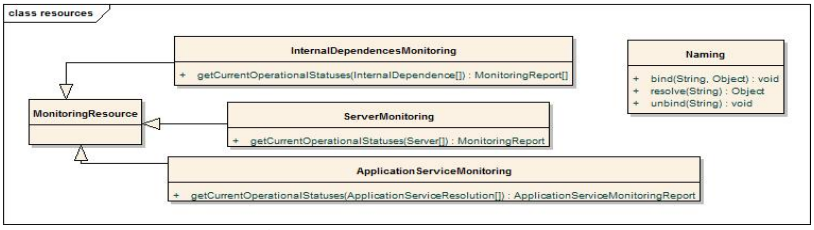

Fig. 5. Monitoring Resources

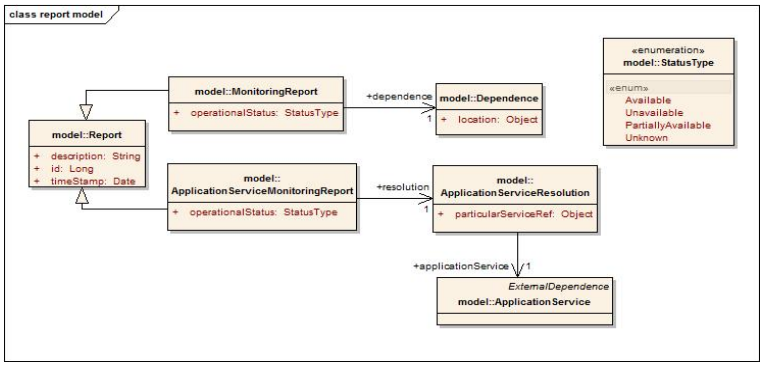

Fig. 6. Reports model

"InternalDependenceMonitoring" resource is in charge of getting the properties values of each managed component's internal dependence, and of inferring its operational status.

The "ServerMonitoring" resource is responsible for monitoring the servers as File servers, Database servers, etc, which are used by the managed components.

The "ApplicationServiceMonitoring" resource is responsible for monitoring application services used by the managed component. It monitors specific services instead of abstract services. It generates reports containing the service resolution of the abstract service dependence.

Monitoring resources generate reports that are read by agents to get the operational status of both internal dependencies of managed components, and external dependencies of those components. The Information about what to monitor is provided by the two XML description files: the Internal Dependence Description File (IDDF), and the External Dependence Description File (EDDF).

Agents use the Resources to monitor and analyze the internal structure of the managed component. The monitoring of external components is performed through queries and publish/subscribe mechanisms. Agents also gather information about the Managed Component state from the "ProxyOfControl". This information is used to achieve fault diagnosis.

3.4 The behaviour of a self-managed component

The computing behaviour of a self-managed component will be illustrated with a working self-repair example taken from the Artist Community system, which has been used for validating the approach. The scenario is based on the failure of one running components –"GestionUsuarios"–, which affects the component "ProcesamientoTrabajos" depending on it. The system behaviour is depicted in figure 7. The Runtime Agent in "ProcesamientoTrabajos" detects the possible malfunction through its monitoring capability.

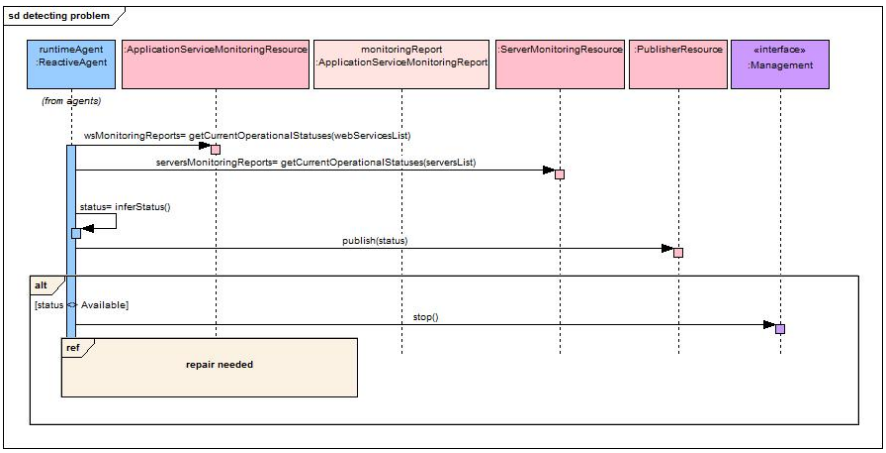

Fig. 7. A repair case

The Runtime Agent publishes the inferred status and stops the Managed Component “ProcesamientoTrabajos” because repair is needed. Then, it requests to the Manager to create an instance of the Repair Agent, which will be in charge of solving the problem. This agent first elaborates a repair plan in order to rebind an alternative of “GestionUsuarios”, and then instantiates and executes the plan.

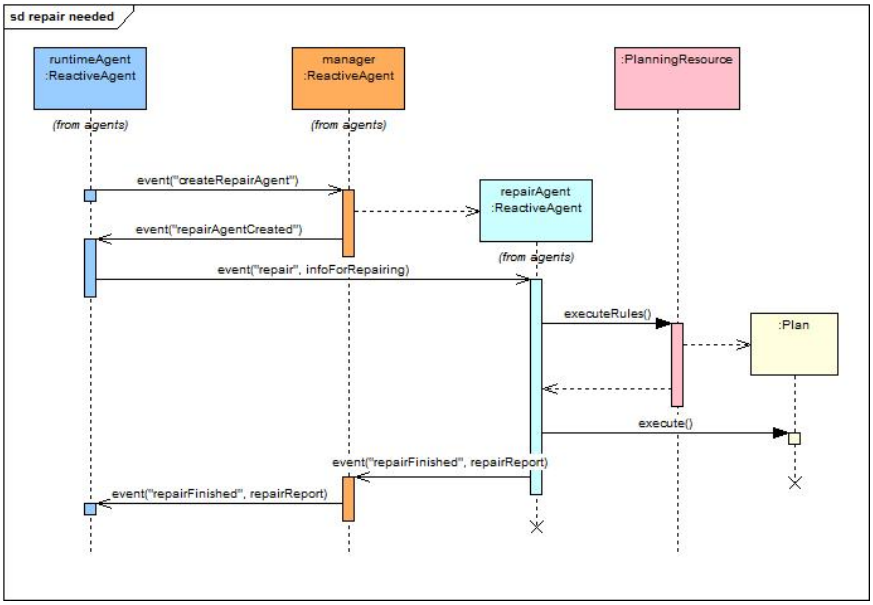

Fig. 8. Creation and execution of a plan by the Repair Agent

The repairing plan successes because an alternative service is available in the system. The new service rebound by the execution of the plan will be monitored in the next cycles of the Runtime Agents. If the new service's status is Available, the Runtime Agent will infer an Available status for the managed component, start it and publish the new status.

4 Planning model

A Plan in this model is a sequence of Tasks. A Task is defined as an operator that somehow changes the environment state pursuing some objective.

The preparation of a plan consists in chaining different tasks in sequence. This process is dynamically performed by an agent anytime it detects some issue reported by monitoring resources with the intention to solve the problem.

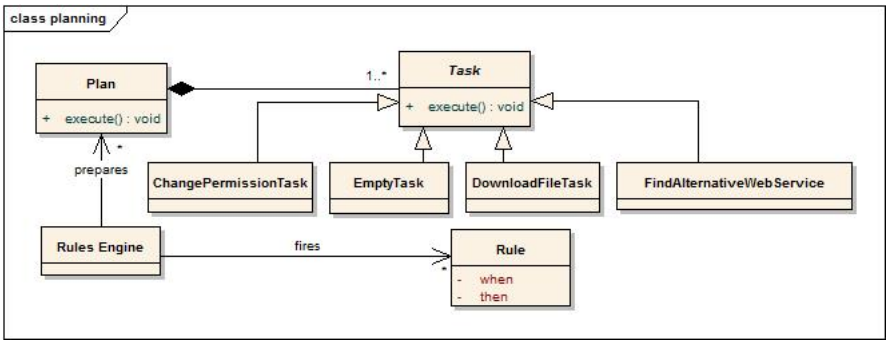

Fig. 9. The Planning Model

For an agent to decide which tasks are included in the plan, a set of "when-then" rules whose "when" part contains the possible symptoms detected for the possible issues. These rules are defined in a text file and fired by a rules engine based on RETE algorithm [9]. A rule example is given in figure 10. The set of predefined tasks and the rules file can be extended in order to customize the agents' behaviour against some issue.

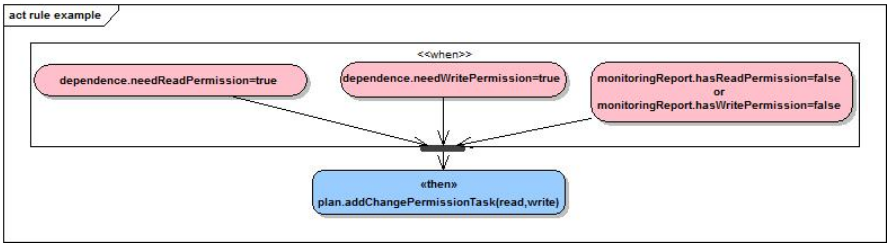

Fig. 10. A rule example

The preparation of the plan is finished when there are no more rules to fire. The plan is then ready to be executed, usually by the agent that prepared it.

5 Validation

The framework has been validated building a distributed system for assisting a Graphic Arts Community (the users) and then enhancing each system component with self-management capabilities.

The system is made of separated components that perform the different functions, some of them requiring others to their own functionality. The system is implemented using Java™ language and JAX-WS framework to support remote access via Web Services technology. Their interfaces and Access Points are registered in a central UDDI Registry. In addition, the system uses a SMTP Server, a Database Server and a UDDI Registry Server.

The transformation framework is made of a set of classes and file resources implemented with Java™, which are included together with business classes to generate a unique deployable component that runs on the same platform.

After framework application, the system has been successfully tested with a collection of significant scenarios including: restoration of missing files, XML validations, rebinding of services with replicated Web Services, etc.

Results showed that, although the computational overload is perceptibly increased, user-system interactions are not affected, while service continuity and stability are significantly improved.

6 Conclusions

The results obtained with the prototype show that the self-managed components perform successfully local monitoring, dynamic plan synthesis, and plan execution for component troubleshooting. Coordination among components is also achieved for fault diagnosis and self-healing. Compared to other approaches based on hierarchical management structures, making each component self-managed enforces their autonomy for failure detection and problem solving, and peer-to peer communication among components provides robustness and fault tolerance at system level. Decentralized control has also well known shortcomings with respect to centralized approaches, as the need of more sophisticated protocols for communication and cooperation. However, for large systems the advantages overcome the disadvantages, because this kind of architecture avoids bottlenecks, are more flexible and can be easily extended.

Future work should focus on self-optimization, self-configuration and self-protection. The last objective could be achieved following a similar approach consisting on enhancing each component with self-protection capabilities. This idea may not be applicable to the first two objectives. Achieving self-optimization and self-configuration would require system-wide parameter adjustment based on global system features that must be obtained seamlessly from system components. Therefore, individual components should "agree" on the proposed changes to achieve the global system behaviour. This might be done through the introduction of component's *choreographies* –group tasks carried out by agents in order to achieve common objectives, which are supported by some interaction protocols.

Another key issue is the automatic generation of Proxy classes and configuration files from code annotations made by developers in the business component code. This might be done by developing specific tools that will interpret the code annotations to detect component's usage of Web Services and other external components, as well as internal dependences. This annotation-oriented dependence declaration style, seems more intuitive and less error-prone than hardcoding dependency description files.

Finally, the self-management framework can be applied to itself since it is also a system. This can be useful to prevent errors in management tasks and to ensure that the machinery (configuration files and auxiliary classes) is ready.

Acknowledgements. We gratefully acknowledge Telefónica I+D, and the INGENIAS group for their help and support to carry out this work.

References

1. Jeffrey O. Kephart and David M. Chess (2003). The Vision of Autonomic Computing. Computer Magazine on January, pp. 41-50
2. Desmond F. D'Souza, Alan C. Wills. Objects, Components and Frameworks With UML. Addison Wesley, 1999.
3. OASIS WSDM Standards, An Introduction to WSDM, http://docs.oasis-open.org/wsdm/wsdm-1.0-intro-primer-cd-01.pdf
4. OASIS WSDM Standards, Management Using Web Services Part 2, http://docs.oasis-open.org/wsdm/wsdm-muws2-1.1-spec-os-01.pdf
5. Ana Mas, Agentes Software y Sistemas Multi-Agentes: Conceptos, Arquitecturas y Aplicaciones, section 2.3
6. Ana Mas, Agentes Software y Sistemas Multi-Agentes: Conceptos, Arquitecturas y Aplicaciones, section 2.2.2
7. Garijo, F J., Bravo S., Gonzalez, J. Bobadilla E. BOGAR_LN: An Agent Based Component Framework for Developing Multi-modal Services using Natural Language. L.N.A. I, Vol 3040. Pp 207-. Springer-Verlag.2004
8. The ICARO-T Framework. Internal report, Telefónica I+D May 2008
9. IBM, An architectural blueprint for autonomic computing, section 2.2
10. Charles Forgy, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence, 19, pp 17-37, 1982
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software
12. IBM, An architectural blueprint for autonomic computing.
13. Jonghyun Lee, Karpjoo Jeong, Hanku Lee, Inho Lee, Sangmoon Lee, Dosik Park, Changsung Lee, Woojin Yang, RISE: A Grid-Based Self-Configuring and Self-Healing Remote System Image Management Environment, Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)
14. Kevin Lee, Rizos Sakellariou, Norman W. Paton and Alvaro A. A. Fernandes, Workflow Adaptation as an Autonomic Computing Problem, WORKS'07
15. P. Martin, W. Powley, K. Wilson2, W. Tian, T. Xu1 and J. Zebedee, The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services, International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)
16. Shameem Ahmed, Sheikh I. Ahamed, Moushumi Sharmin, and Munirul M. Haque, Self-healing for Autonomic Pervasive Computing, SAC'07